Forest Automata for Shape Analysis

Peter Habermehl[1], Ondřej Lengál[3], Lukáš Holík[2,3], Adam Rogalewicz[3], Jiří Šimáček[3,4], and **Tomáš Vojnar**[3]

[1]LIAFA, Université Paris Diderot—Paris 7/CNRS, France

[2]Department of Information Technology, Uppsala University, Sweden

[3]**FIT, Brno University of Technology, Czech Republic**

[4]VERIMAG, UJF/CNRS/INPG, Gières, France

January 29, 2013

Programs with Pointers

- Programs using pointers and dynamic linked data structures (lists, trees, ...) are often hard to get right and understand, bugs in them are hard to find.

- For example, consider the following code

```
for(pos = ((typeof(*pos) *)((char *)((&hl_list)->next)
          -(unsigned long)(&((typeof(*pos) *)0)->head)));
    &pos->head != (&gl_list);
    pos = ((typeof(*pos) *)((char *)(pos->head.next)
          -(unsigned long)(&((typeof(*pos) *)0)->head))))
  { printf(" %d", pos->value); }
```

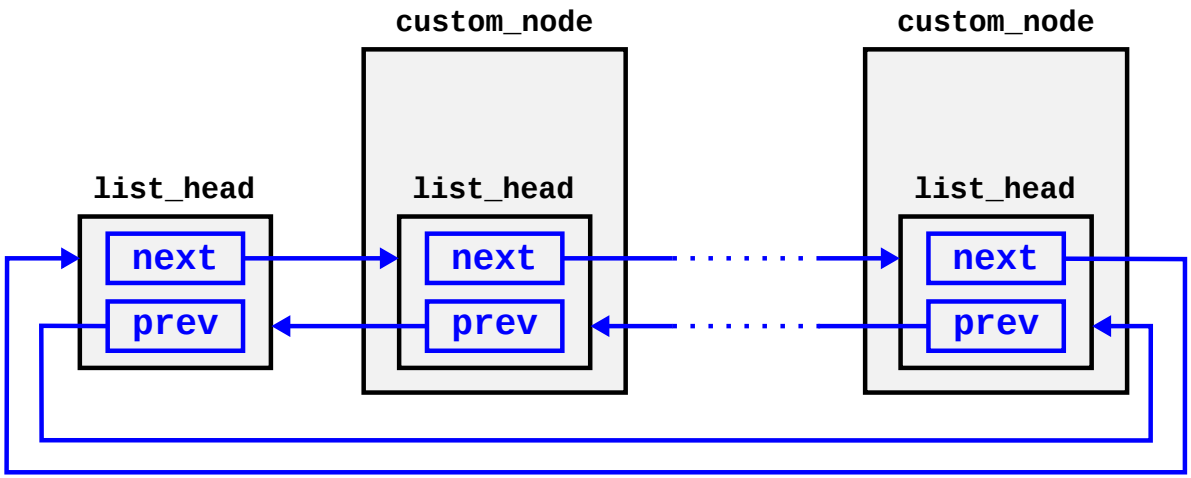
Programs with Pointers

- Programs using pointers and dynamic linked data structures (lists, trees, ...) are often hard to get right and understand, bugs in them are hard to find.

- For example, consider the following code

```
for(pos = ((typeof(*pos) *)((char *)((&hl_list)->next)
          -(unsigned long)(&((typeof(*pos) *)0)->head)));
    &pos->head != (&gl_list);
    pos = ((typeof(*pos) *)((char *)(pos->head.next)
          -(unsigned long)(&((typeof(*pos) *)0)->head))))
  { printf(" %d", pos->value); }
```

implementing a Linux list traversal (after macro expansion):



Analysing Programs with Pointers

- Programs with pointers and dynamic data structures are also hard to analyse due to their infinite state-spaces consisting of graphs of unrestricted shape and size.
  - Usually undecidable.
  - Even reasonable semi-algorithmic solutions, i.e.,
    - no guarantee of termination of the analysis,
    - possibility of false positives/negatives or "don't know" answers, and/or
    - human help,

    are not easy.

Properties To Be Checked

- Basic memory safety:
  - Absence of invalid memory address dereferencing: no null/undefined pointer dereferences.
  - Absence of double free operations.
  - Garbage freeness: all dynamically allocated blocks are reachable from program variables (no memory leaks).
- Shape invariants: heap configurations that a program can generate (at a certain line) have a desired shape—e.g., they indeed correspond to unshared acyclic lists, trees, etc.
- Checking assertions built into the code by programmers.

- Various data-related properties: sortedness, preservation of the contents, ...
- Termination or even complexity of algorithms.

The last two items are not considered in the presented approach.

Common Ideas of Many Shape Analyses

- Use a suitable formalism to finitely encode infinite sets of reachable heap configurations:
  - various kinds of finite (word, tree, forest, ...) automata,
  - various logics (3-valued predicate logic with transitive closure, MSO on trees/graphs, WSkS, separation logic, ...),
  - graphs with summary nodes, graph grammars, ...
  - ... possibly extended with various numerical and other constraints: length, sortedness, data properties, termination, ...

Common Ideas of Many Shape Analyses

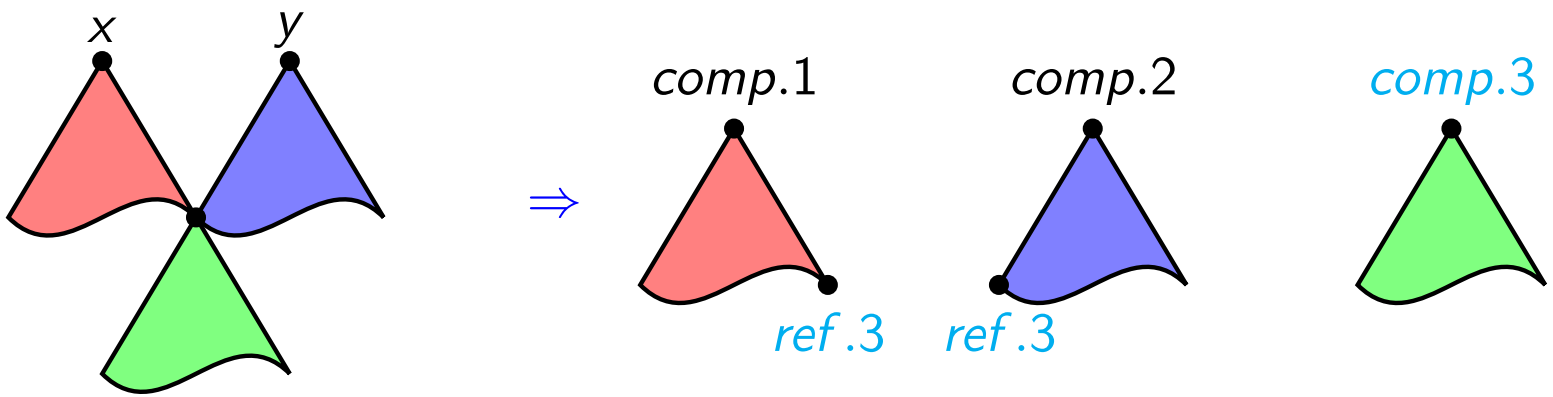
- Use a suitable formalism to finitely encode infinite sets of reachable heap configurations:
  - various kinds of finite (word, tree, forest, ...) automata,
  - various logics (3-valued predicate logic with transitive closure, MSO on trees/graphs, WSkS, separation logic, ...),
  - graphs with summary nodes, graph grammars, ...
  - ... possibly extended with various numerical and other constraints: length, sortedness, data properties, termination, ...
- Evaluate programs statements symbolically over sets of program configurations encoded using the chosen formalism.

Common Ideas of Many Shape Analyses

- Use a suitable formalism to finitely encode infinite sets of reachable heap configurations:
  - various kinds of finite (word, tree, forest, ...) automata,
  - various logics (3-valued predicate logic with transitive closure, MSO on trees/graphs, WSkS, separation logic, ...),
  - graphs with summary nodes, graph grammars, ...
  - ... possibly extended with various numerical and other constraints: length, sortedness, data properties, termination, ...
- Evaluate programs statements symbolically over sets of program configurations encoded using the chosen formalism.
- Computing (finitely-represented) infinite sets of reachable configurations by simply iterating program statements does typically not converge!
  - Break a program into loop-free fragments by providing loop invariants.
  - Accelerate the computation, e.g., by using abstraction, to jump to the fix-point.

Heaps as Forests

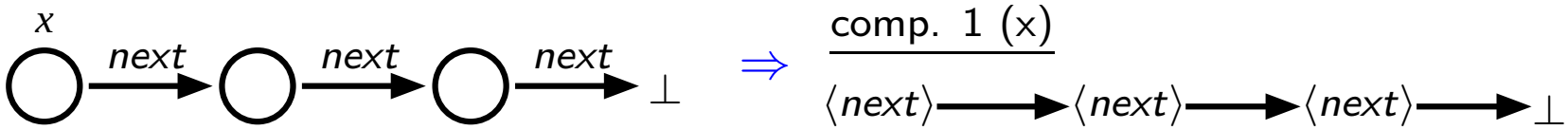
- We encode a heap as a forest = a finite tuple of trees + a variable mapping.
- Each node of a heap which is pointed multiple times or by a program variable becomes a root of one tree component.
- The tree components refer to each other via explicit root references:



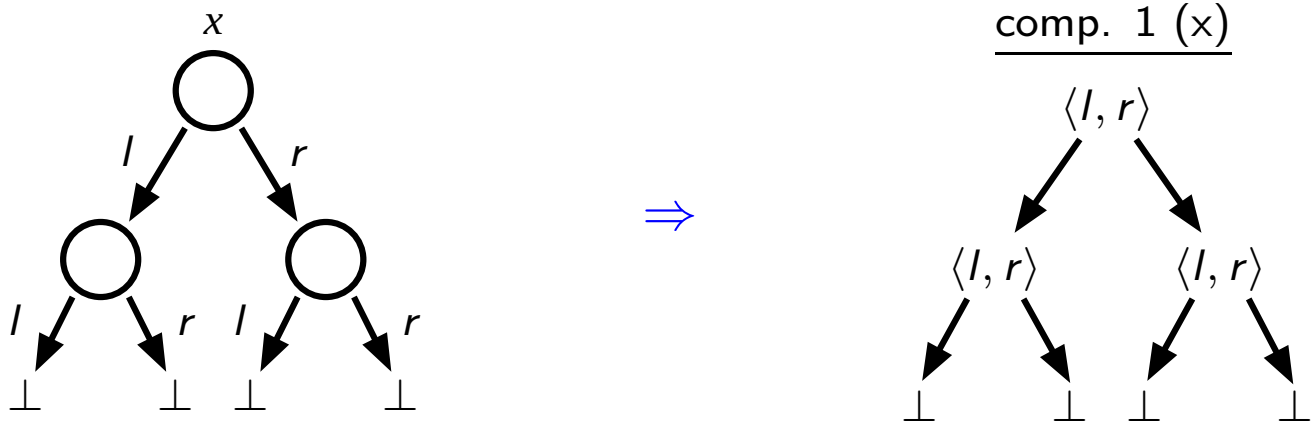
Variable mapping = $\{x \to 1, y \to 2\}$

Examples of Forest Encoding I

- A singly-linked list:

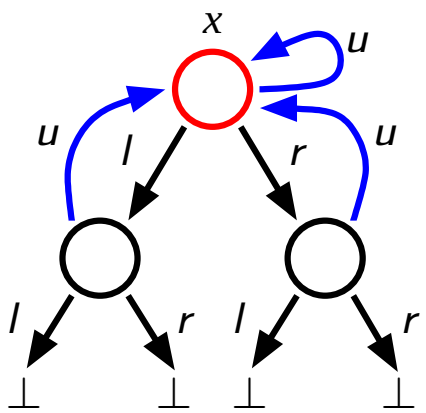


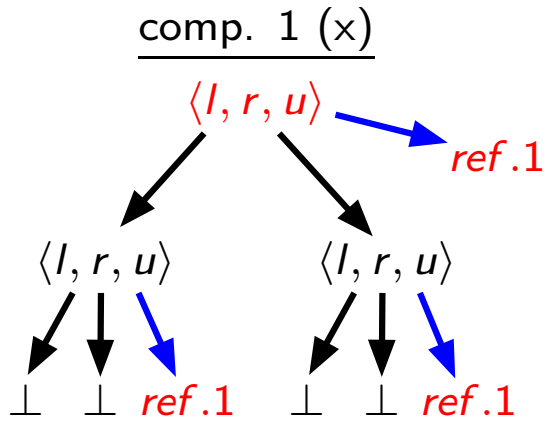
- A tree:



Examples of Forest Encoding II

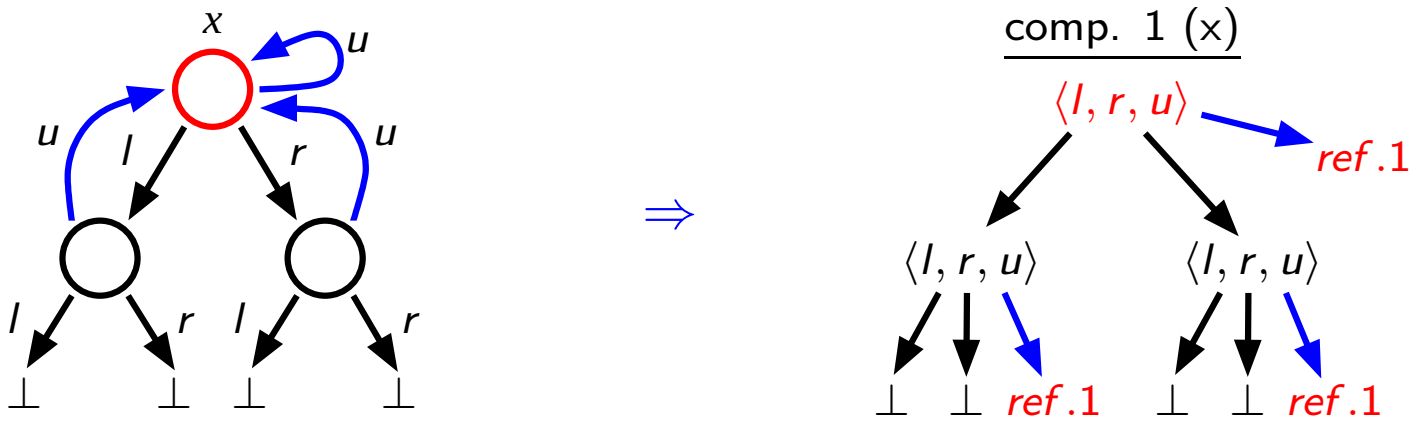
- A tree with root pointers:



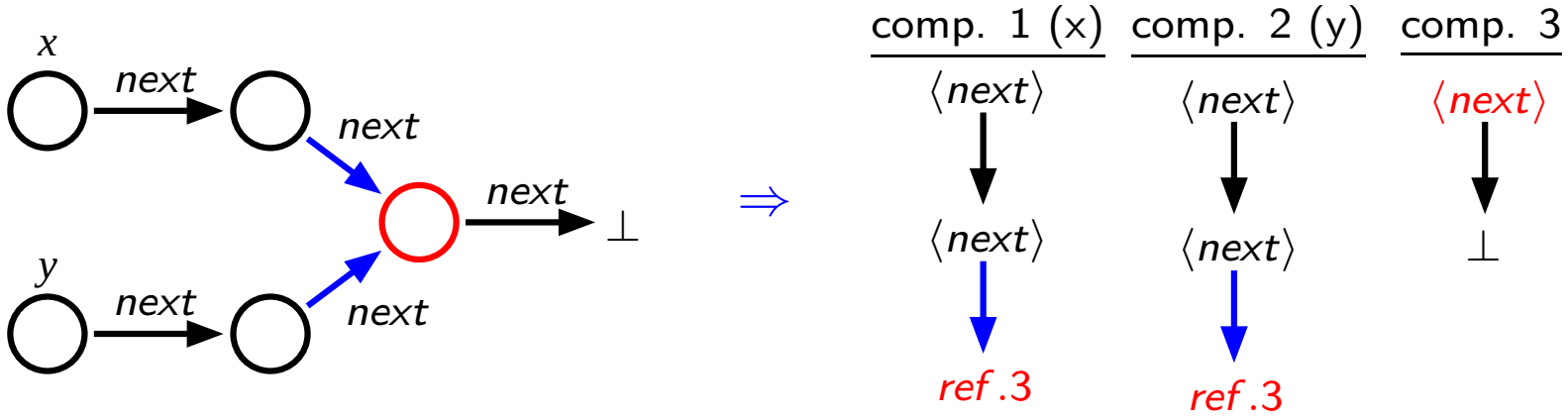


Examples of Forest Encoding II

- A tree with root pointers:



- Shared singly-linked lists:

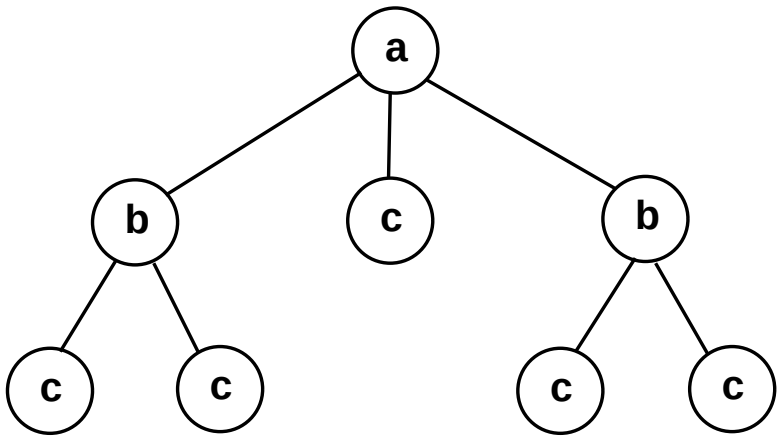


Top-Down Tree Automata

- A top-down tree automaton (TA): $\mathcal{A} = (Q, \Sigma, q_0, \Delta)$ where
  - Q is a finite set of states,
  - Σ is a finite ranked alphabet with a rank function $\# : \Sigma \to \mathbb{N}_0$,
  - $q_0 \in Q$ is the initial state,
  - Δ is a set of tree transition rules of the form $q \xrightarrow{a} (q_1, ..., q_n)$ where $q, q_1, ..., q_n \in Q$ and $a \in \Sigma$ s.t. $\#(a) = n$.

- An example of a run of a TA accepting a tree:

$$\Delta = \{ \quad q_0 \xrightarrow{a} (q_1, q_2, q_1), \quad q_1 \xrightarrow{b} (q_2, q_2), \quad q_2 \xrightarrow{c} () \quad \}$$



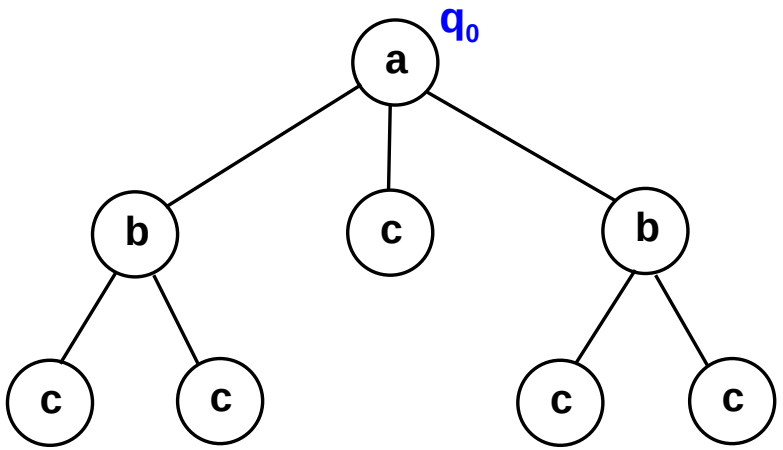
- The language $L(\mathcal{A})$ of a TA $\mathcal{A}$ consists of all the trees it accepts.

Top-Down Tree Automata

- A top-down tree automaton (TA): $\mathcal{A} = (Q, \Sigma, q_0, \Delta)$ where
  - Q is a finite set of states,
  - Σ is a finite ranked alphabet with a rank function $\# : \Sigma \to \mathbb{N}_0$,
  - $q_0 \in Q$ is the initial state,
  - Δ is a set of tree transition rules of the form $q \xrightarrow{a} (q_1, ..., q_n)$ where $q, q_1, ..., q_n \in Q$ and $a \in \Sigma$ s.t. $\#(a) = n$.

- An example of a run of a TA accepting a tree:

$$\Delta = \{ \quad q_0 \xrightarrow{a} (q_1, q_2, q_1), \quad q_1 \xrightarrow{b} (q_2, q_2), \quad q_2 \xrightarrow{c} () \quad \}$$



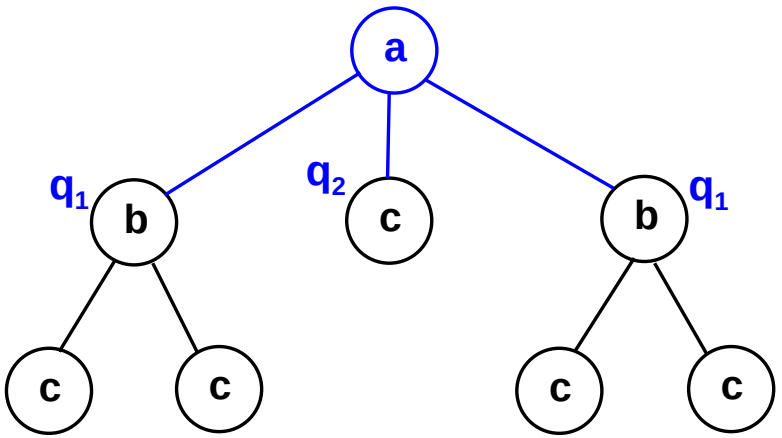
- The language $L(\mathcal{A})$ of a TA $\mathcal{A}$ consists of all the trees it accepts.

Top-Down Tree Automata

- A top-down tree automaton (TA): $\mathcal{A} = (Q, \Sigma, q_0, \Delta)$ where
  - Q is a finite set of states,
  - Σ is a finite ranked alphabet with a rank function $\# : \Sigma \to \mathbb{N}_0$,
  - $q_0 \in Q$ is the initial state,
  - Δ is a set of tree transition rules of the form $q \xrightarrow{a} (q_1, ..., q_n)$ where $q, q_1, ..., q_n \in Q$ and $a \in \Sigma$ s.t. $\#(a) = n$.

- An example of a run of a TA accepting a tree:

$$\Delta = \{ \quad q_0 \xrightarrow{a} (q_1, q_2, q_1), \quad q_1 \xrightarrow{b} (q_2, q_2), \quad q_2 \xrightarrow{c} () \quad \}$$



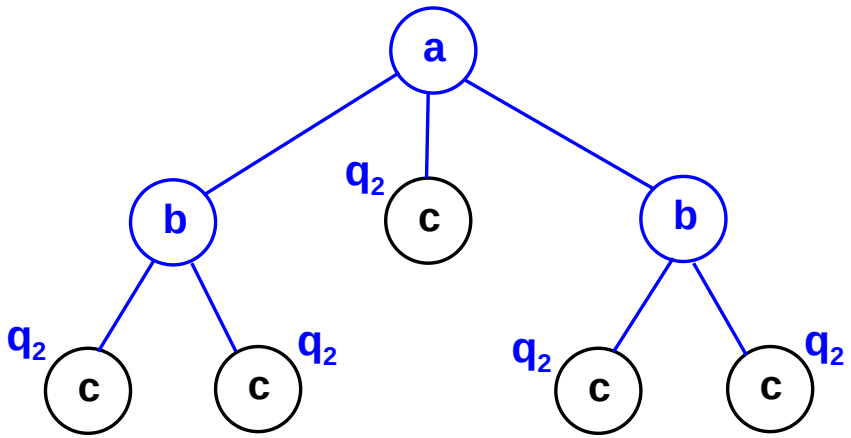
- The language $L(\mathcal{A})$ of a TA $\mathcal{A}$ consists of all the trees it accepts.

Top-Down Tree Automata

- A top-down tree automaton (TA): $\mathcal{A} = (Q, \Sigma, q_0, \Delta)$ where
  - Q is a finite set of states,
  - Σ is a finite ranked alphabet with a rank function $\# : \Sigma \to \mathbb{N}_0$,
  - $q_0 \in Q$ is the initial state,
  - Δ is a set of tree transition rules of the form $q \xrightarrow{a} (q_1, ..., q_n)$ where $q, q_1, ..., q_n \in Q$ and $a \in \Sigma$ s.t. $\#(a) = n$.

- An example of a run of a TA accepting a tree:

$$\Delta = \{ \quad q_0 \xrightarrow{a} (q_1, q_2, q_1), \quad q_1 \xrightarrow{b} (q_2, q_2), \quad q_2 \xrightarrow{c} () \quad \}$$



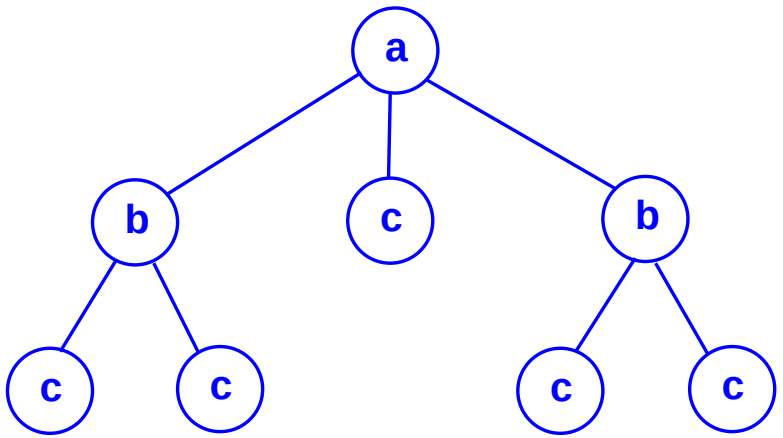
- The language $L(\mathcal{A})$ of a TA $\mathcal{A}$ consists of all the trees it accepts.

Top-Down Tree Automata

- A top-down tree automaton (TA): $\mathcal{A} = (Q, \Sigma, q_0, \Delta)$ where
  - Q is a finite set of states,
  - Σ is a finite ranked alphabet with a rank function $\# : \Sigma \to \mathbb{N}_0$,
  - $q_0 \in Q$ is the initial state,
  - Δ is a set of tree transition rules of the form $q \xrightarrow{a} (q_1, ..., q_n)$ where $q, q_1, ..., q_n \in Q$ and $a \in \Sigma$ s.t. $\#(a) = n$.

- An example of a run of a TA accepting a tree:

$$\Delta = \{ \quad q_0 \xrightarrow{a} (q_1, q_2, q_1), \quad q_1 \xrightarrow{b} (q_2, q_2), \quad q_2 \xrightarrow{c} () \quad \}$$



- The language $L(\mathcal{A})$ of a TA $\mathcal{A}$ consists of all the trees it accepts.

Forest Automata

- A forest automaton (FA) $\mathcal{F} = ((\mathcal{A}_1, \ldots, \mathcal{A}_n), V)$ consists of a tuple of TA and a variable mapping.

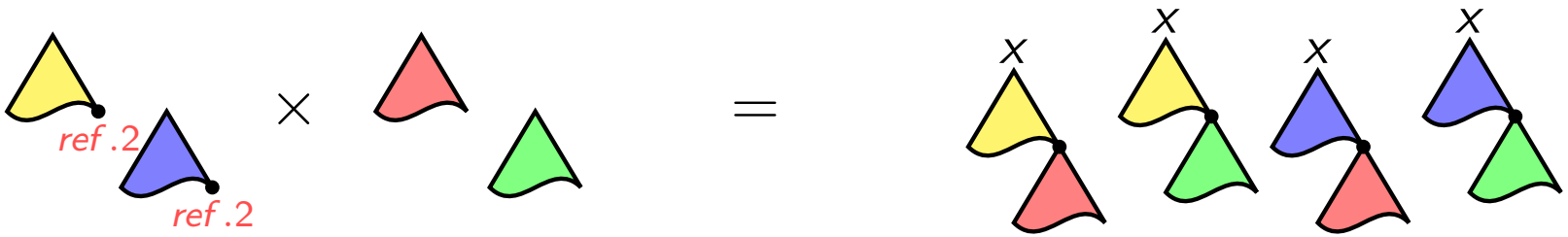
- $\mathcal{F}$ represents the set of heaps

$$\{heap((t_1, \ldots, t_n), V) \mid (t_1, \ldots, t_n) \in L(\mathcal{A}_1) \times \cdots \times L(\mathcal{A}_n)\}$$

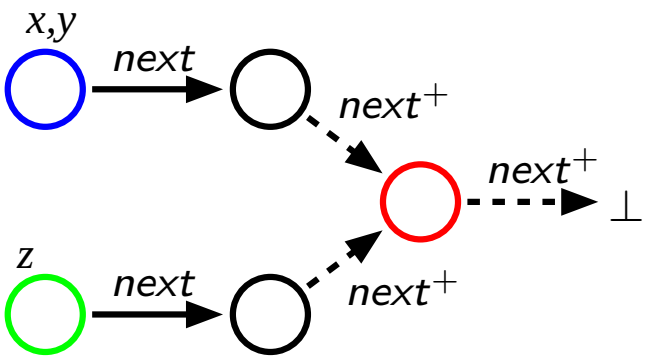
where $heap((t_1, \ldots, t_n), V)$ denotes the heap obtained by gluing $t_1, ..., t_n$ through the root references.

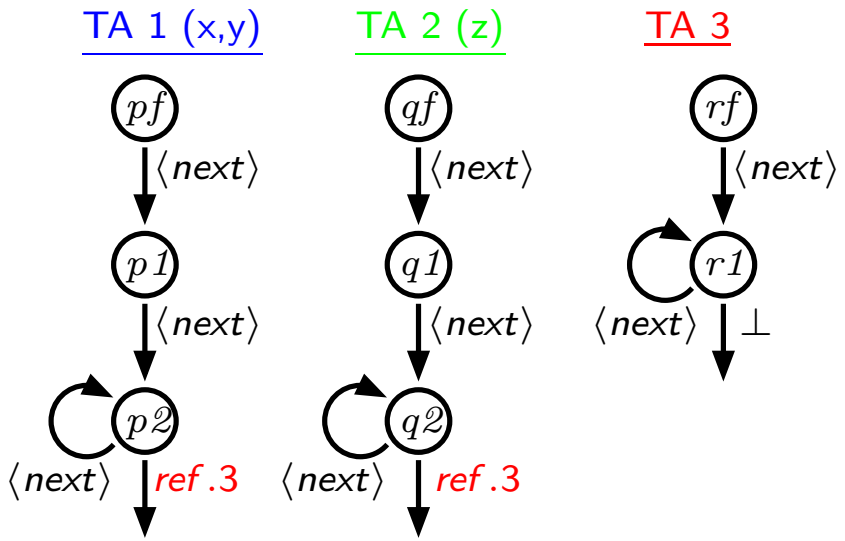
- For example:

$$(\ (\ \mathcal{A}_1 \quad , \quad \mathcal{A}_2\ ), \quad \{x \to 1\}\ )$$

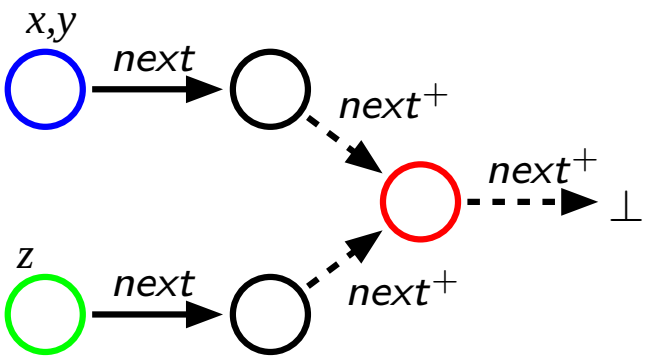


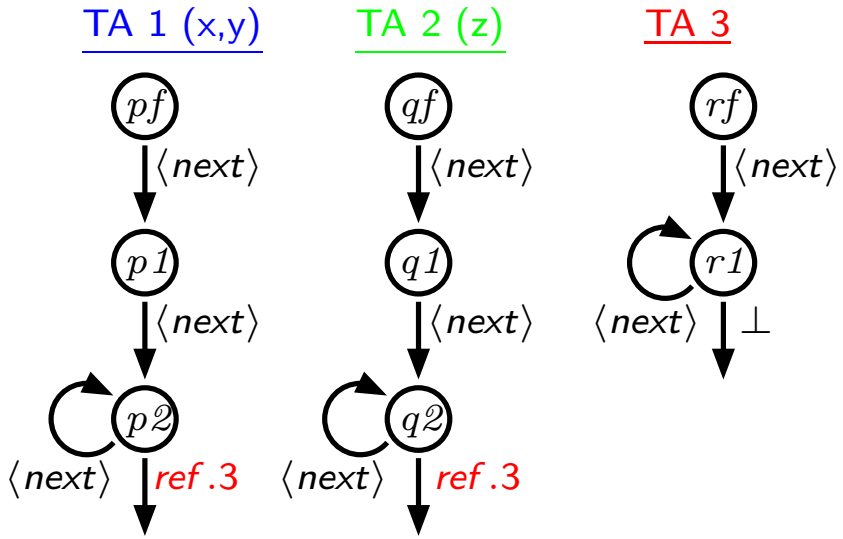
Pointer Updates





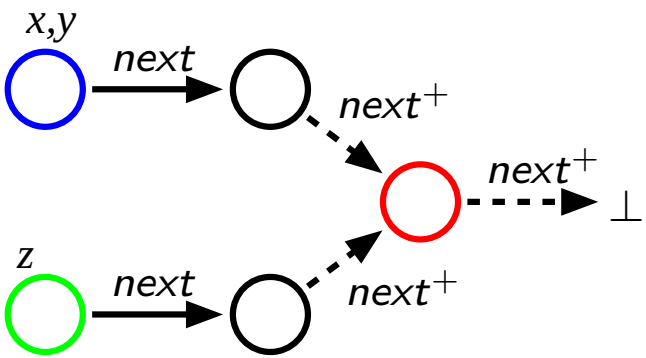
Pointer Updates

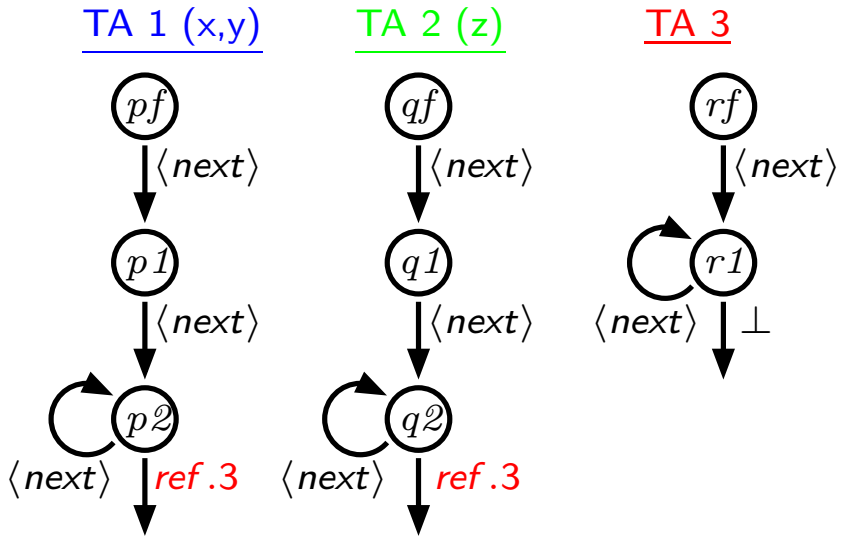




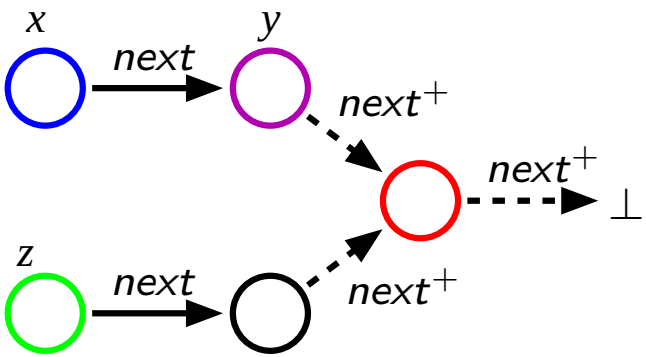
- y:=x.next

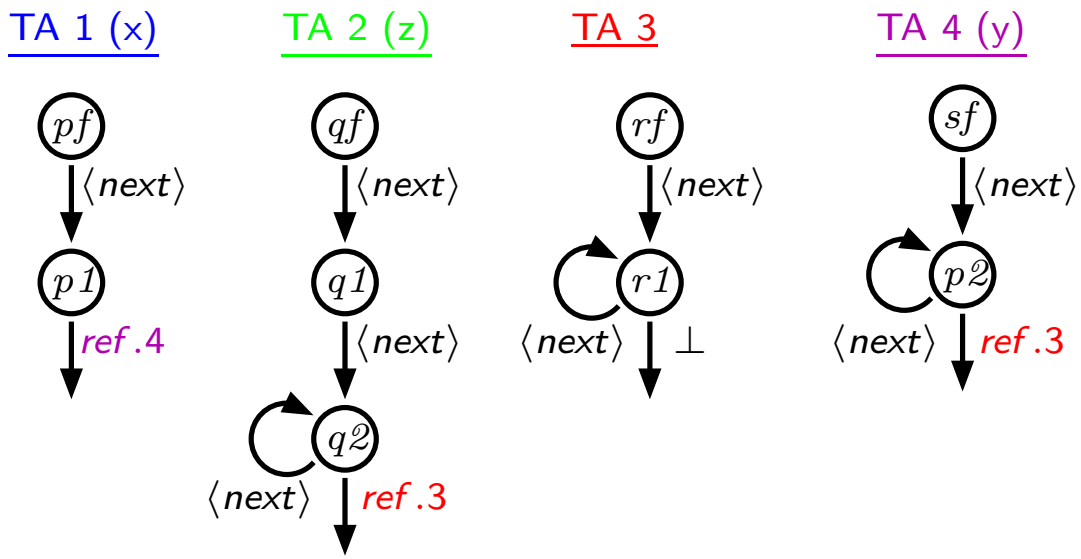
Pointer Updates



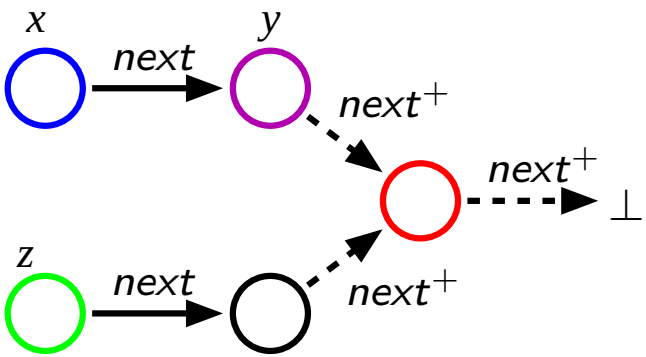


- y:=x.next

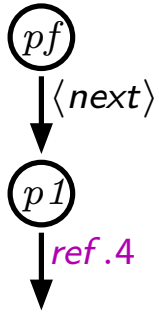




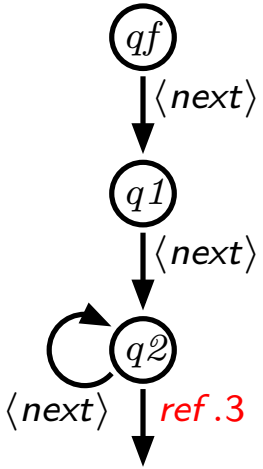
Pointer Updates



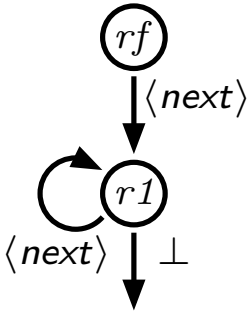
TA 1 (x)



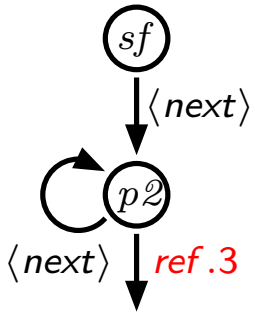
TA 2 (z)



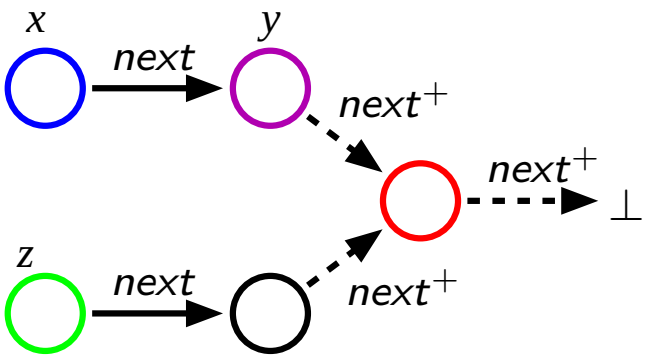
TA 3



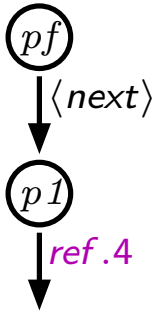
TA 4 (y)



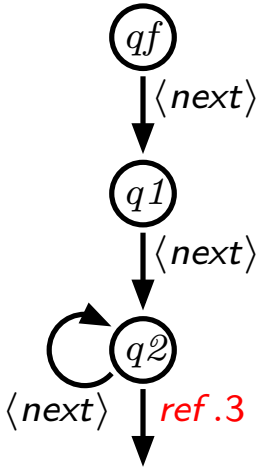
Pointer Updates



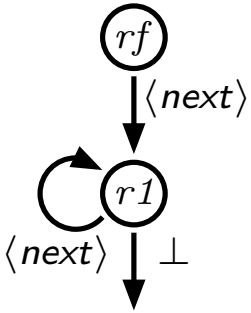
TA 1 (x)



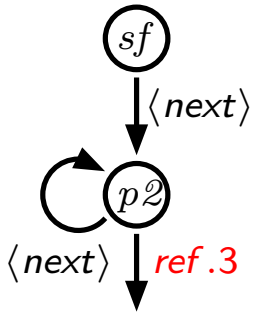
TA 2 (z)



TA 3

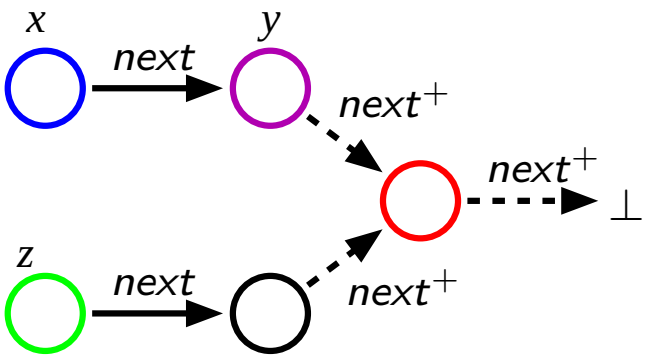


TA 4 (y)

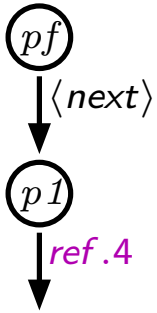


- x.next:=z;

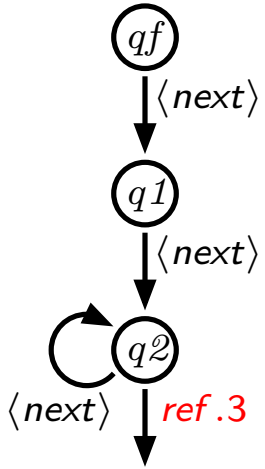
Pointer Updates



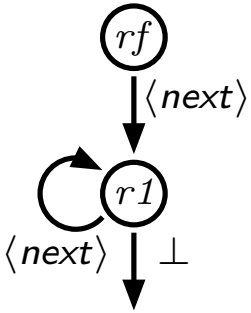
TA 1 (x)



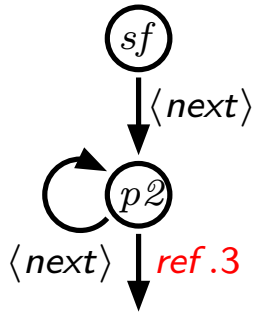
TA 2 (z)



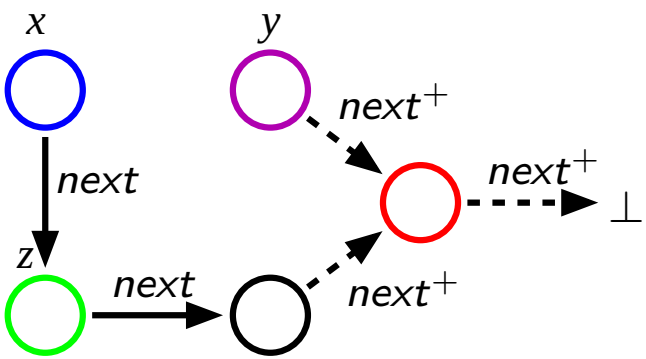
TA 3



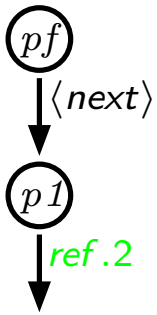
TA 4 (y)



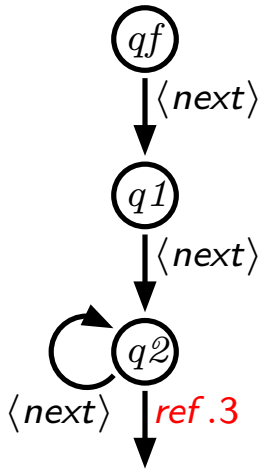
- x.next:=z;



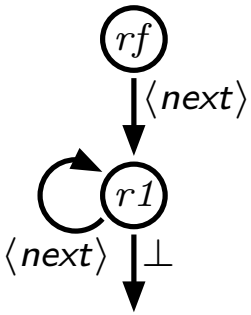
TA 1 (x)



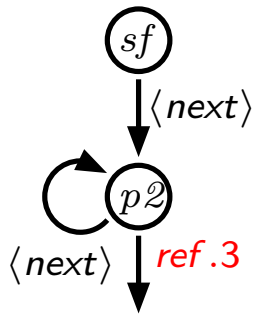
TA 2 (z)



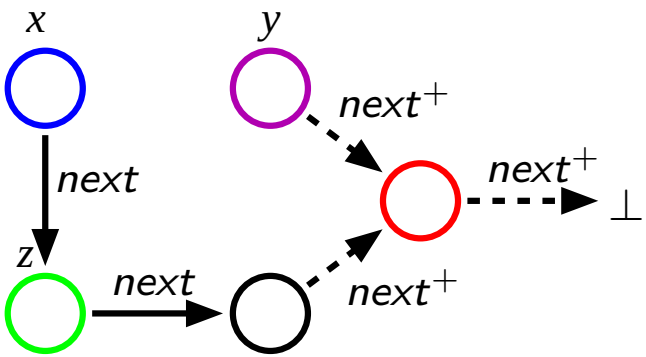
TA 3



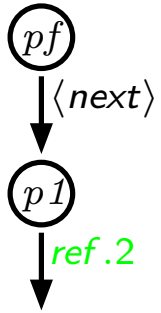
TA 4 (y)



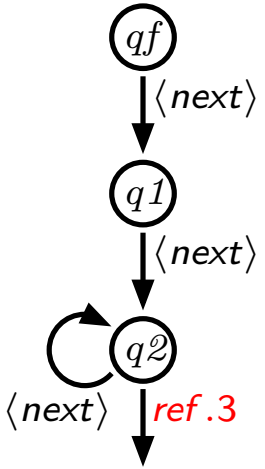
Pointer Updates



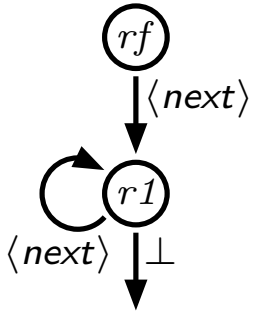
TA 1 (x)



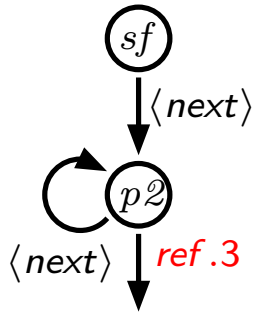
TA 2 (z)



TA 3

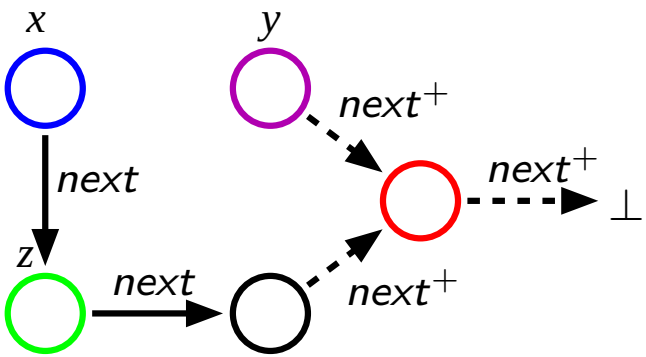


TA 4 (y)

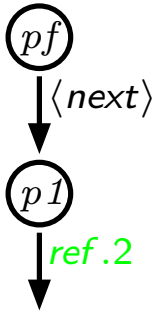


- z:=x;

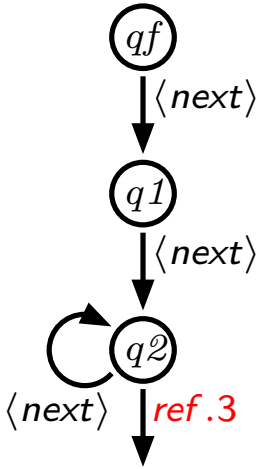
Pointer Updates



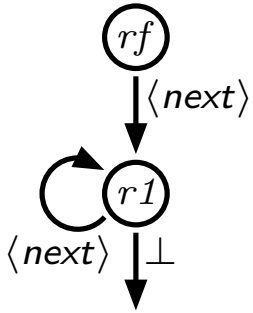
TA 1 (x)



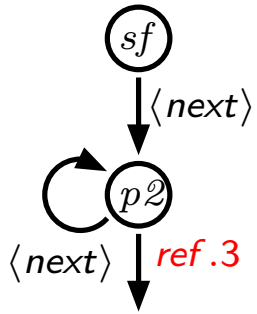
TA 2 (z)



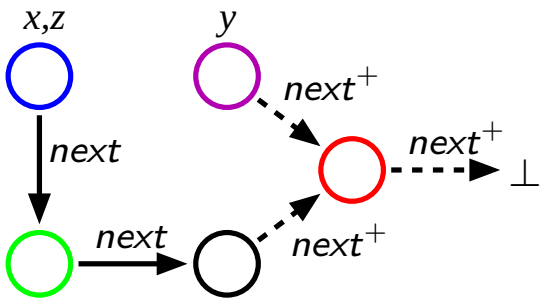
TA 3



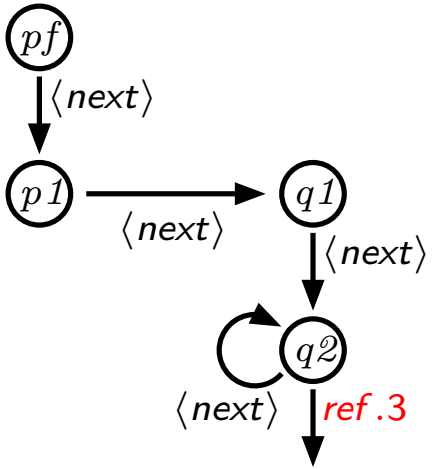
TA 4 (y)



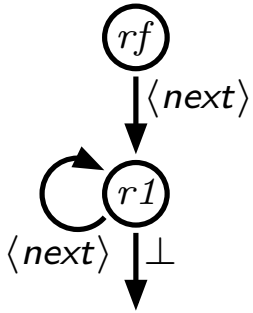
- z:=x;



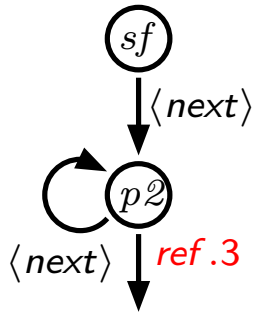
TA 1 (x,z)



TA 3



TA 4 (y)



Deciding Entailment

- A forest $((t_1, \ldots, t_n), V)$ is canonical iff
  - every root is referenced either by a program variable or more than once,
  - the trees are ordered according to the order of discovering their roots in a sequence of depth-first traversals through $heap((t_1, \ldots, t_n), V)$ that starts at program variables in a predefined variable-order.
  - Then, two canonical forests encode the same graph iff they are identical.

Deciding Entailment

- A forest $((t_1, \ldots, t_n), V)$ is canonical iff
  - every root is referenced either by a program variable or more than once,
  - the trees are ordered according to the order of discovering their roots in a sequence of depth-first traversals through $heap((t_1, \ldots, t_n), V)$ that starts at program variables in a predefined variable-order.
  - Then, two canonical forests encode the same graph iff they are identical.

- A forest automaton $((\mathcal{A}_1, \ldots, \mathcal{A}_n), V)$ is canonical (CFA) iff every forest $((t_1, \ldots, t_n), V)$, $(t_1, \ldots, t_n) \in L(\mathcal{A}_1) \times \cdots \times L(\mathcal{A}_n)$ is canonical.

Deciding Entailment

- A forest $((t_1, \ldots, t_n), V)$ is canonical iff
  - every root is referenced either by a program variable or more than once,
  - the trees are ordered according to the order of discovering their roots in a sequence of depth-first traversals through $heap((t_1, \ldots, t_n), V)$ that starts at program variables in a predefined variable-order.
  - Then, two canonical forests encode the same graph iff they are identical.

- A forest automaton $((\mathcal{A}_1, \ldots, \mathcal{A}_n), V)$ is canonical (CFA) iff every forest $((t_1, \ldots, t_n), V)$, $(t_1, \ldots, t_n) \in L(\mathcal{A}_1) \times \cdots \times L(\mathcal{A}_n)$ is canonical.

- Entailment of CFAs reduces to tree automata language inclusion:
  For two CFAs $\mathcal{F} = ((\mathcal{A}_1, \ldots, \mathcal{A}_n), V)$ and $\mathcal{F}' = ((\mathcal{A}'_1, \ldots, \mathcal{A}'_m), V')$,

  graphs encoded by $\mathcal{F}$ are a subset of those encoded by $\mathcal{F}'$
  if and only if
  $n = m$, $V = V'$, and $L(\mathcal{A}_1) \subseteq L(\mathcal{A}'_1), \ldots, L(\mathcal{A}_n) \subseteq L(\mathcal{A}'_n)$.

Deciding Entailment

- A forest $((t_1, \ldots, t_n), V)$ is canonical iff
  - every root is referenced either by a program variable or more than once,
  - the trees are ordered according to the order of discovering their roots in a sequence of depth-first traversals through $heap((t_1, \ldots, t_n), V)$ that starts at program variables in a predefined variable-order.
  - Then, two canonical forests encode the same graph iff they are identical.

- A forest automaton $((\mathcal{A}_1, \ldots, \mathcal{A}_n), V)$ is canonical (CFA) iff every forest $((t_1, \ldots, t_n), V)$, $(t_1, \ldots, t_n) \in L(\mathcal{A}_1) \times \cdots \times L(\mathcal{A}_n)$ is canonical.

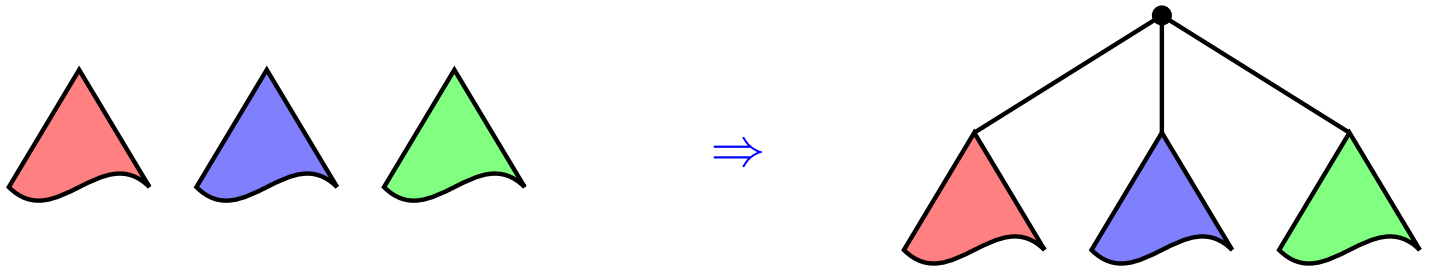
- Entailment of CFAs reduces to tree automata language inclusion:
  For two CFAs $\mathcal{F} = ((\mathcal{A}_1, \ldots, \mathcal{A}_n), V)$ and $\mathcal{F}' = ((\mathcal{A}'_1, \ldots, \mathcal{A}'_m), V')$,

  graphs encoded by $\mathcal{F}$ are a subset of those encoded by $\mathcal{F}'$
  if and only if
  $n = m$, $V = V'$, and $L(\mathcal{A}_1) \subseteq L(\mathcal{A}'_1), \ldots, L(\mathcal{A}_n) \subseteq L(\mathcal{A}'_n)$.

- Any FA can be transformed into a finite set of CFAs.

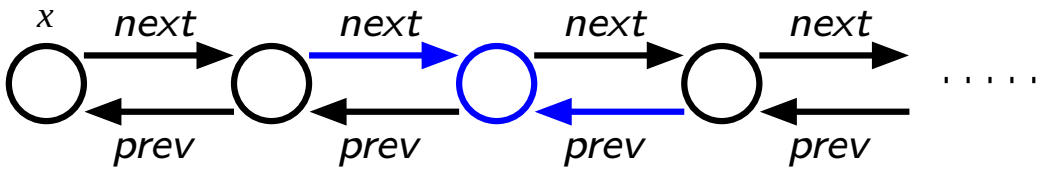
Deciding Entailment II

- Inclusion between sets of CFAs can be reduced to inclusion of ordinary TAs.
- One can convert a tuple of TAs into a single TA by adding a designated node on top of each tuple of trees.



Hierarchical Forest Automata

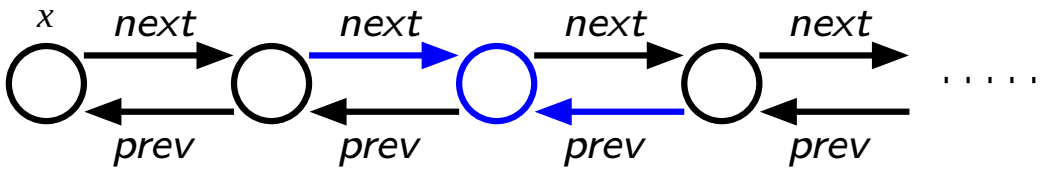
- In a doubly-linked list, each node is a root-point (two incoming edges).



We have to deal with an unbounded number of root-points.

Hierarchical Forest Automata

- In a doubly-linked list, each node is a root-point (two incoming edges).

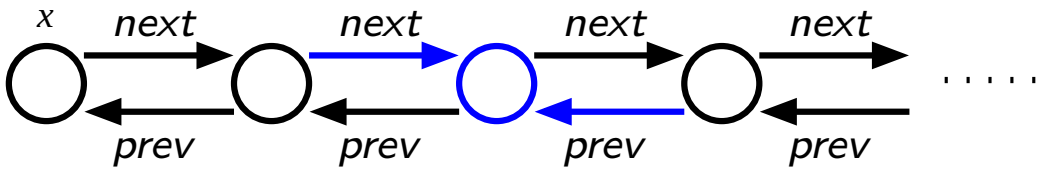


  We have to deal with an unbounded number of root-points.

- To handle such structures, we propose a hierarchical encoding.

Hierarchical Forest Automata

- In a doubly-linked list, each node is a root-point (two incoming edges).

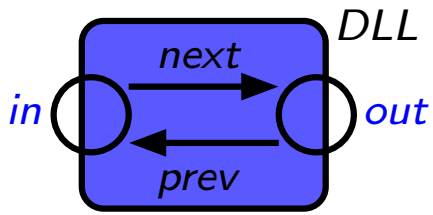


  We have to deal with an unbounded number of root-points.

- To handle such structures, we propose a hierarchical encoding.

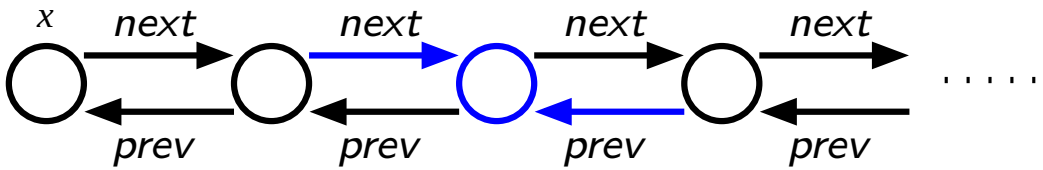
- We close a pair of doubly-linked nodes into a box:
  - a sort of a predicate described itself as an FA.

  Boxes can be used as alphabet symbols of other FA.



Hierarchical Forest Automata

- In a doubly-linked list, each node is a root-point (two incoming edges).

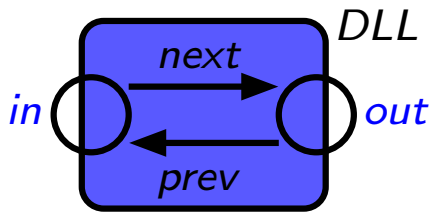


  We have to deal with an unbounded number of root-points.

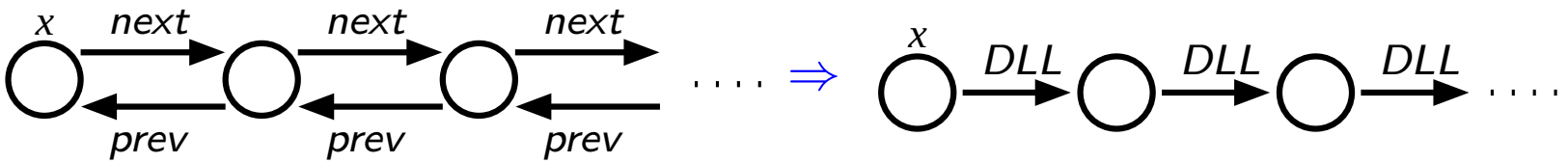
- To handle such structures, we propose a hierarchical encoding.

- We close a pair of doubly-linked nodes into a box:
  - a sort of a predicate described itself as an FA.

  Boxes can be used as alphabet symbols of other FA.



- Hence we get:



Hierarchical Forest Automata II

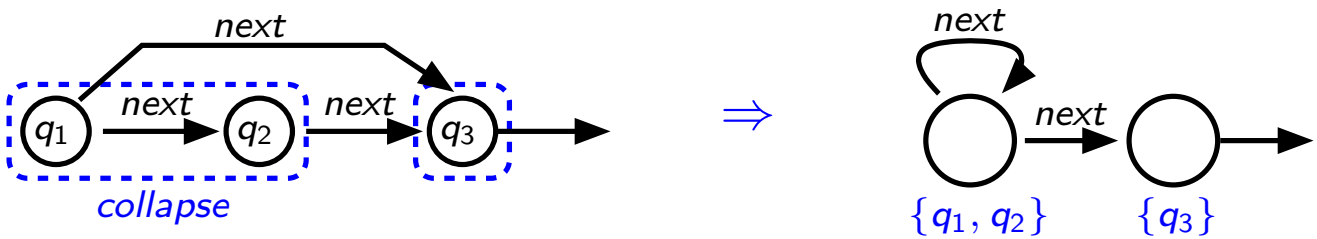
- Boxes allow us to handle structures like trees with parent-pointers, (doubly-linked) lists of cyclic lists, skip-lists, etc.

Hierarchical Forest Automata II

- Boxes allow us to handle structures like trees with parent-pointers, (doubly-linked) lists of cyclic lists, skip-lists, etc.
- They bring complications too:
  - The symbolic pointer manipulation needs to fold and unfold boxes.
  - The boxes either need to be provided manually, or they have to be learnt by looking for suitable sub-graphs to be folded.
    - Trying to reduce the in-degree of some root-point, using not too big nor too small sub-graphs with a single entry point.
  - Our entailment procedure only safely approximates the entailment.
    - Precise when dealing with maximally "boxed" heaps using uniquely named boxes that do not overlap (often the case in practice).

Abstraction

- To achieve termination, we use abstractions introduced in abstract regular model checking based on collapsing states of an automaton whose languages are considered similar.
- So far we build on the simple bounded height abstraction that collapses states that accept the same trees up to some height.



- The abstraction is applied to each component of an FA separately, several FA-specific optimizations (both refinement as well as coarsening) are used.

Verification Procedure

- A program is executed symbolically on CFAs trying to reach an error configuration or a fix-point given by a set of CFAs for every program location.
- Usual pointer manipulating statements are supported, together with some restricted pointer arithmetic:
  `x := malloc(<size>)`, `x := null`, `x := y`, `x := y.next`, `x.next := y`, `if/while (x = y)`, `free(x)`.
- Box discovery and folding and a transformation into the CFA form are performed at every step, abstraction only when several program paths meet.
- An important ingredient for efficiency is the use of nondeterministic TA:
  - never determinise,
  - use antichain-based TA inclusion checking for fix-point testing,
  - use simulation-based reduction to keep automata small.

An Experimental Evaluation Using a Tool Called Forester

Example	Forester (sec)	boxes	Predator (sec)
SLL (delete)	0.04		0.04
SLL (bubblesort)	0.04		0.03
SLL (mergesort)	0.15		0.10
SLL (insertsort)	0.05		0.04
SLL (reverse)	0.03		0.03
SLL+head	0.05		0.03
SLL of 0/1 SLLs	0.03		0.11
$\mathrm{SLL}_{\mathrm{Linux}}$	0.03		0.03
SLL of CSLLs	0.73	3 / 4	0.12
SLL of $\mathrm{2CDLLs}_{\mathrm{Linux}}$	0.17	13 / 5	0.25
skip list_2	0.42	- / 3	T
skip list_3	9.14	- / 7	T
DLL (reverse)	0.06	1 / 1	0.03
DLL (insert)	0.07	1 / 1	0.05
DLL (insertsort1)	0.40	1 / 1	0.11
DLL (insertsort2)	0.12	1 / 1	0.05
DLL of CDLLs	1.25	8 / 7	0.22
DLL+subdata	0.09	- / 2	T
CDLL	0.03	1 / 1	0.03
tree	0.14		Err
tree+parents	0.21	2 / 2	T
tree+stack	0.08		Err
tree (DSW)	0.40		Err
tree of CSLLs	0.42	- / 4	Err

Summary and Future Work

- A concept of forest automata and a shape analysis based on them were presented.
- The analysis is fully-automated and both quite general as well as scalable.
- Implemented in a gcc plug-in called Forester – freely available (together with the twin tool Predator and many case studies), see the tools at:

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

- Still, a number of directions for future work remain:
  - A proper investigation and description of the box learning approach.
  - More advanced abstraction with automatic refinement.
  - Support for non-pointer data, concurrency, recursive boxes, recursion, ...