Efficient Algorithms for Using Automata in Infinite-State Verification

Ondřej Lengál

Advisor: prof. Ing. Tomáš Vojnar, Ph.D.

Faculty of Information Technology
Brno University of Technology

May 28, 2014

Scope of the Thesis

Formal verification of programs with complex dynamic data structures,

- e.g. lists, trees (**red-black**), skip lists, ...
- used in OS kernels, standard libraries, concurrent libraries, ...

using the formal theory of automata,

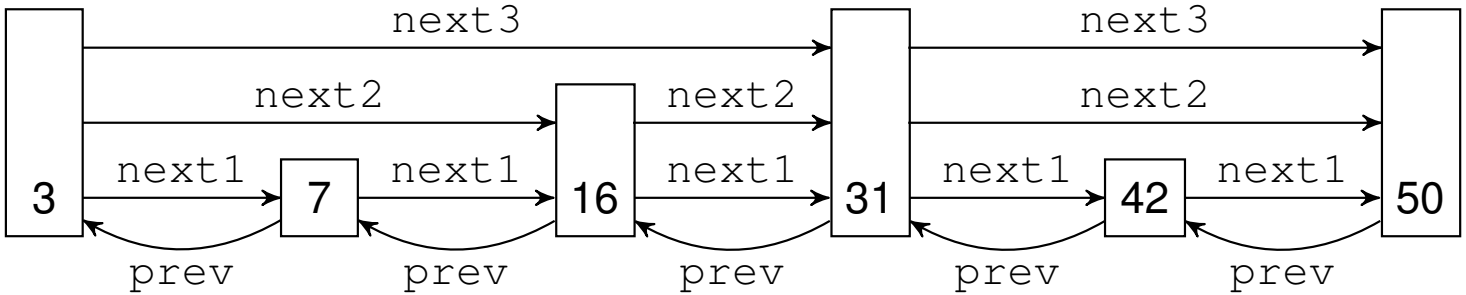
- $\rightsquigarrow$ development of efficient automata manipulation techniques,

and the use of automata in other applications,

- $\rightsquigarrow$ decision procedures of logics: WSkS, separation logic.

Motivating Example

- H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-threaded systems. J. Parallel Distrib. Comput. 65.
- A lock-free implementation of a highly concurrent skip-list.



Motivating Example

- Procedure for insertion of a new value:

```
function Insert(key:integer, value:pointer to Value):
 boolean
I1      TraverseTimeStamps();
I2    Choose level randomly according to the skip list
        distribution
I3    newNode:=CreateNode(level,key,value);
I4    COPY_NODE(newNode);
I5    node1:=COPY_ NODE(head);
I6    for i:=maxLevel-1 to 1 step -1 do
I7      node2:=ScanKey(&node1,i,key);
I8      RELEASE_NODE(node2);
I9      if i< level then savedNodes[i]:=
          COPY_ NODE(node1);
I10   while true do
I11     node2:=ScanKey(&node1,0,key);
I12     ⟨value2,d⟩:=node2.value;
I13     if d=false and node2.key=key then
I14       if CAS(&node2.value,⟨value2,false⟩,
            ⟨value,false⟩) then
I15         RELEASE_NODE(node1);
I16         RELEASE_NODE(node2);
I17         for i:=1 to level-1 do
I18           RELEASE_NODE(savedNodes[i]);
I19         RELEASE_NODE(newNode);
I20         RELEASE_NODE(newNode);
I21         return true2;
I22       else
I23         RELEASE_NODE(node2);
I24         continue;
I25     newNode.next[0]:= ⟨node2,false⟩,
I26     RELEASE_NODE(node2);
I27     if CAS(&node1.next[0],⟨node2,false⟩,
          ⟨newNode,false⟩) then
I28       RELEASE_NODE(node1);
I29       break;
I30     Back-Off
I31   for i:=1 to level-1 do
I32     newNode.validLevel:=i;
I33     node1:=savedNodes[i];
I34     while true do
I35       node2:=ScanKey(&node1,i,key);
I36       newNode.next[i]:=⟨node2,false⟩;
I37       RELEASE_NODE(node2);
I38       if newNode.value.d=true or
            CAS(&node1.next[i],node2,newNode) then
I39         RELEASE_NODE(node1);
I40         break;
I41       Back-Off
I42   newNode.validLevel:=level;
I43     newNode.timeInsert:=getNextTimeStamp();
I44   if newNode.value.d=true then newNode:=
        HelpDelete(newNode,0);
I45   RELEASE_ NODE(newNode);
I46   return true;
```

Motivating Example

- Procedure for insertion of a new value:

```
function Insert(key:integer, value:pointer to Value):
 boolean
I1      TraverseTimeStamps();
I2    Choose level randomly according to the skip list
        distribution
I3    newNode:=CreateNode(level,key,value);
I4    COPY_NODE(newNode);
I5    node1:=COPY_ NODE(head);
I6    for i:=maxLevel-1 to 1 step -1 do
I7      node2:=ScanKey(&node1,i,key);
I8      RELEASE_NODE(node2);
I9      if i< level then savedNodes[i]:=
          COPY_ NODE(node1);
I10   while true do
I11     node2:=ScanKey(&node1,0,key);
I12     ⟨value2,d⟩:=node2.value;
I13     if d=false and node2.key=key then
I14       if CAS(&node2.value,⟨value2,false⟩,
            ⟨value,false⟩) then
I15         RELEASE_NODE(node1);
I16         RELEASE_NODE(node2);
I17         for i:=1 to level-1 do
I18           RELEASE_NODE(savedNodes[i]);
I19         RELEASE_NODE(newNode);
I20         RELEASE_NODE(newNode);
I21         return true2;
I22       else
I23         RELEASE_NODE(node2);
I24         continue;
I25     newNode.next[0]:= ⟨node2,false⟩,
I26     RELEASE_NODE(node2);
I27     if CAS(&node1.next[0],⟨node2,false⟩,
          ⟨newNode,false⟩) then
I28       RELEASE_NODE(node1);
I29       break;
I30     Back-Off
I31   for i:=1 to level-1 do
I32     newNode.validLevel:=i;
I33     node1:=savedNodes[i];
I34     while true do
I35       node2:=ScanKey(&node1,i,key);
I36       newNode.next[i]:=⟨node2,false⟩;
I37       RELEASE_NODE(node2);
I38       if newNode.value.d=true or
            CAS(&node1.next[i],node2,newNode) then
I39         RELEASE_NODE(node1);
I40         break;
I41       Back-Off
I42   newNode.validLevel:=level;
I43     newNode.timeInsert:=getNextTimeStamp();
I44   if newNode.value.d=true then newNode:=
        HelpDelete(newNode,0);
I45   RELEASE_ NODE(newNode);
I46   return true;
```

- Is the procedure correct?
  - i.e. linearizable, lock-free, shape-invariant, memory-safe, . . .
  - difficult to prove/validate $\rightsquigarrow$ **automated techniques desired**!

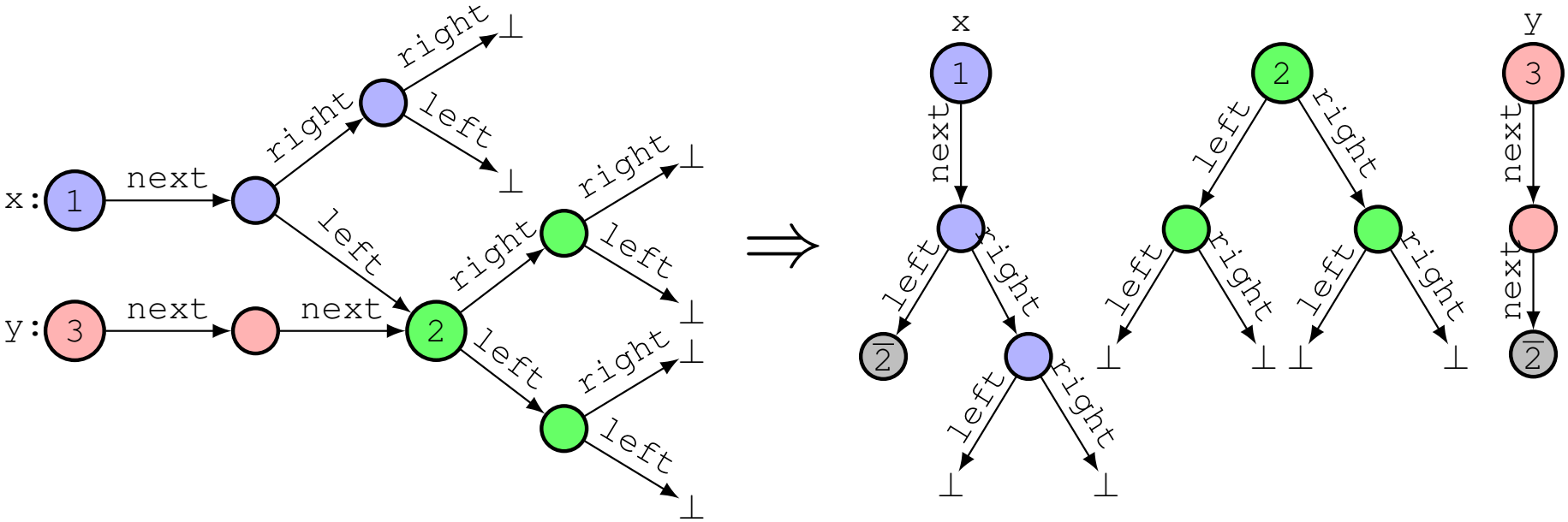
Forest Automata-based Verification

- Verification of memory-safety of heap-manipulating programs,
- infinite-state programs $\rightsquigarrow$ needs to represent heap symbolically,
- representation mostly based on logics, graphs, automata.

Forest Automata-based Verification

Our approach:

- decompose heap into tree components (a forest)

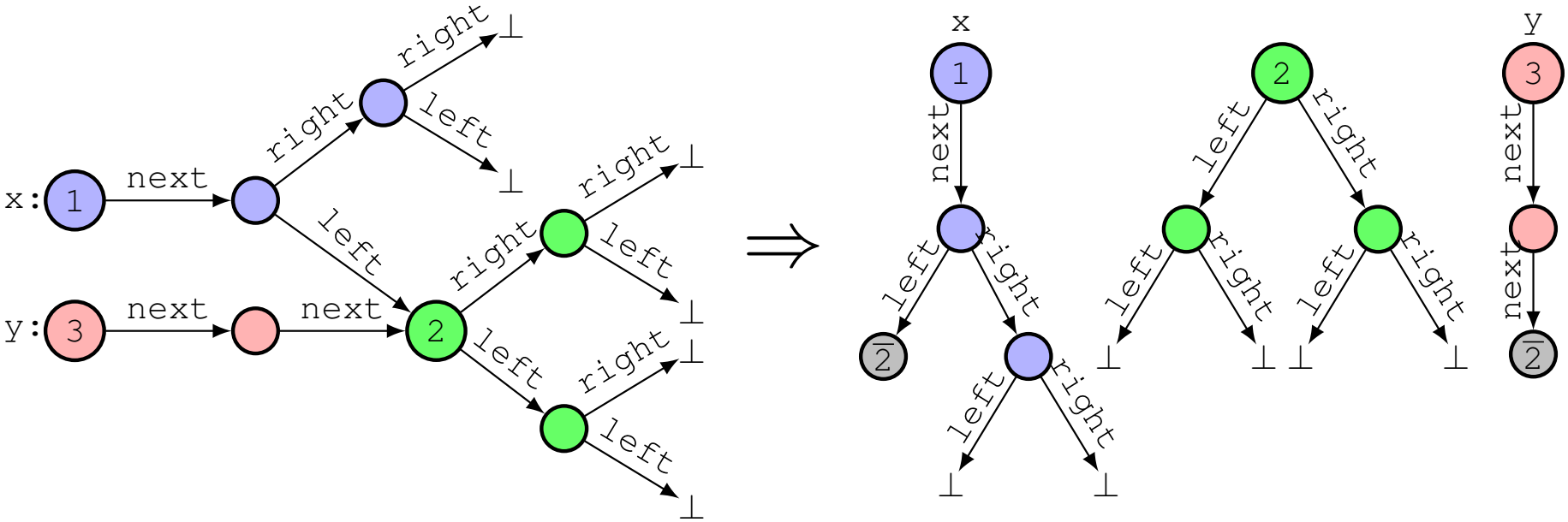


a) a graph, and b) its **forest** representation

Forest Automata-based Verification

Our approach:

- decompose heap into tree components (a forest)



a) a graph, and b) its **forest** representation

- sets of heaps:
  - collect 1st, 2nd, . . . trees from all forests into sets of trees,
  - represent each set of trees by a tree automaton,
  - a tuple of tree automata $\rightsquigarrow$ a **forest automaton**.

Forest Automata-based Verification

The analysis:

- based on abstract interpretation:
- for every line of code, compute forest automata representing reachable heap configurations at this line, until fixpoint,
- program statements are substituted by abstract transformers performing the corresponding operation on forest automata,
- at loop points, do widening (over-approximation),
- hierarchical forest automata:
  - **box** — a forest automaton used as a symbol,
  - necessary for more complex data structures (DLLs, skip lists, . . .).

Forest Automata-based Verification

- Fully automated shape analysis with forest automata
  - finding the right boxes is hard,
  - developed an algorithm to find suitable subgraphs,
  - works for a large class of data structures (including skip lists),
  - Holík, Lengál, Rogalewicz, Šimáček, and Vojnar. Fully Automated Shape Analysis Based on Forest Automata. In *Proc. of CAV'13*, LNCS 8044.

Forest Automata-based Verification

- Fully automated shape analysis with forest automata
  - finding the right boxes is hard,
  - developed an algorithm to find suitable subgraphs,
  - works for a large class of data structures (including skip lists),
  - Holík, Lengál, Rogalewicz, Šimáček, and Vojnar. Fully Automated Shape Analysis Based on Forest Automata. In *Proc. of CAV'13*, LNCS 8044.

- Verification of heap programs with ordered data
  - extension of forest automata with ordering relations,
  - verification of sorting algorithms, binary search trees, skip lists, ...
  - Abdulla, Holík, Jonsson, Lengál, Trinh, and Vojnar. Verification of Heap Manipulating Programs with Ordered Data by Extended FA. In *Proc. of ATVA'13*, LNCS 8172.

Forest Automata-based Verification

- Fully automated shape analysis with forest automata
  - finding the right boxes is hard,
  - developed an algorithm to find suitable subgraphs,
  - works for a large class of data structures (including skip lists),
  - Holík, Lengál, Rogalewicz, Šimáček, and Vojnar. Fully Automated Shape Analysis Based on Forest Automata. In *Proc. of CAV'13*, LNCS 8044.

- Verification of heap programs with ordered data
  - extension of forest automata with ordering relations,
  - verification of sorting algorithms, binary search trees, skip lists, ...
  - Abdulla, Holík, Jonsson, Lengál, Trinh, and Vojnar. Verification of Heap Manipulating Programs with Ordered Data by Extended FA. In *Proc. of ATVA'13*, LNCS 8172.

- Verification of programs with highly-concurrent data structures
  - based on a generalization of the thread-modular approach,
  - a goal — fully automatically verify a concurrent skip list,
  - a work in progress.

Efficient Algorithms for Manipulating Automata

The need to efficiently manipulate non-deterministic automata:

- even for language inclusion checking, size reduction, ...

Efficient Algorithms for Manipulating Automata

The need to efficiently manipulate non-deterministic automata:

- even for language inclusion checking, size reduction, ...
- A new tree automata inclusion checking algorithm,
  - traverses the automata downwards,
  - in many cases superior to other algorithms,
  - Holík, Lengál, Šimáček, and Vojnar. Efficient Inclusion Checking on Explicit and Semi-Symbolic TA. In *Proc. of ATVA'11*, LNCS 6996.

Efficient Algorithms for Manipulating Automata

The need to efficiently manipulate non-deterministic automata:

- even for language inclusion checking, size reduction, ...
- A new tree automata inclusion checking algorithm,
  - traverses the automata downwards,
  - in many cases superior to other algorithms,
  - Holík, Lengál, Šimáček, and Vojnar. Efficient Inclusion Checking on Explicit and Semi-Symbolic TA. In *Proc. of ATVA'11*, LNCS 6996.
- A highly efficient library for non-deterministic automata,
  - implementation of state-of-the-art algorithms,
  - being used by a number of researchers,
  - Lengál, Šimáček, and Vojnar. VATA: A Library for Efficient Manipulation of Non-Deterministic TA. In *Proc. of TACAS'12*, LNCS 7214.

Efficient Algorithms for Manipulating Automata

The need to efficiently manipulate non-deterministic automata:

- even for language inclusion checking, size reduction, ...
- A new tree automata inclusion checking algorithm,
  - traverses the automata downwards,
  - in many cases superior to other algorithms,
  - Holík, Lengál, Šimáček, and Vojnar. Efficient Inclusion Checking on Explicit and Semi-Symbolic TA. In *Proc. of ATVA'11*, LNCS 6996.
- A highly efficient library for non-deterministic automata,
  - implementation of state-of-the-art algorithms,
  - being used by a number of researchers,
  - Lengál, Šimáček, and Vojnar. VATA: A Library for Efficient Manipulation of Non-Deterministic TA. In *Proc. of TACAS'12*, LNCS 7214.
- Development of techniques for symbolic automata,
  - usable e.g. in decision procedures of some logics.

Automata-based Decision Procedures for Logics

- Checking entailment $\psi \stackrel{?}{\models} \varphi$ of Separation Logic formulae
  - an alternative way to verify programs with dynamic memory,
  - transforms the problem to checking membership in tree automata,
  - Enea, Lengál, Sighireanu, T. Vojnar. Compositional Entailment Checking for a Fragment of Separation Logic. A work in progress.

Automata-based Decision Procedures for Logics

- Checking entailment $\psi \stackrel{?}{\models} \varphi$ of Separation Logic formulae
  - an alternative way to verify programs with dynamic memory,
  - transforms the problem to checking membership in tree automata,
  - Enea, Lengál, Sighireanu, T. Vojnar. Compositional Entailment Checking for a Fragment of Separation Logic. A work in progress.

- Deciding WSkS formulae,
  - current decision procedures based on deterministic automata,
  - every quantifier alternation yields complementation,
  - $\rightsquigarrow$ exponential blow-up,
  - proposed a procedure based on non-deterministic automata,
  - avoids full-scale determinization,
  - a work in progress.

Questions?