Verification of Heap Manipulating Programs with Ordered Data by Extended Forest Automata

FIT BUT Technical Report Series

Parosh Aziz Abdulla, Lukáš Holík, Bengt Jonsson, Ondřej Lengál, Cong Quy Trinh, and Tomáš Vojnar

Technical Report No. FIT-TR-2013-02
Faculty of Information Technology, Brno University of Technology

Last modified: July 7, 2013

NOTE: This technical report contains an extended version of the ATVA'13 paper with the same name.

Verification of Heap Manipulating Programs with Ordered Data by Extended Forest Automata

Parosh Aziz Abdulla[1], Lukáš Holík[2], Bengt Jonsson[1], Ondřej Lengál[2], Cong Quy Trinh[1], and Tomáš Vojnar[2]

[1] Department of Information Technology, Uppsala University, Sweden
[2] FIT, Brno University of Technology, IT4Innovations Centre of Excellence, Czech Republic

Abstract. We present a general framework for verifying programs with complex dynamic linked data structures whose correctness depends on ordering relations between stored data values. The underlying formalism of our framework is that of forest automata (FA), which has previously been developed for verification of heap-manipulating programs. We extend FA by constraints between data elements associated with nodes of the heaps represented by FA, and we present extended versions of all operations needed for using the extended FA in a fully-automated verification approach, based on abstract interpretation. We have implemented our approach as an extension of the Forester tool and successfully applied it to a number of programs dealing with data structures such as various forms of singly- and doubly-linked lists, binary search trees, as well as skip lists.

1 Introduction

Automated verification of programs that manipulate complex dynamic linked data structures is one of the most challenging problems in software verification. The problem becomes even more challenging when program correctness depends on relationships between data values that are stored in the dynamically allocated structures. Such ordering relations on data are central for the operation of many data structures such as search trees, priority queues (based, e.g., on skip lists), key-value stores, or for the correctness of programs that perform sorting and searching, etc. The challenge for automated verification of such programs is to handle *both* infinite sets of reachable heap configurations that have a form of complex graphs *and* the different possible relationships between data values embedded in such graphs, needed, e.g., to establish sortedness properties.

As discussed below in the section on related work, there exist many automated verification techniques, based on different kinds of logics, automata, graphs, or grammars, that handle dynamically allocated pointer structures. Most of these approaches abstract from properties of data stored in dynamically allocated memory cells. The few approaches that can automatically reason about data properties are often limited to specific classes of structures, mostly singly-linked lists (SLLs), and/or are not fully automated (as also discussed in the related work paragraph).

In this paper, we present a general framework for verifying programs with complex dynamic linked data structures whose correctness depends on relations between the stored data values. Our framework is based on the notion of *forest automata* (FA) which has previously been developed for representing sets of reachable configurations of programs with complex dynamic linked data structures [10,11]. In the FA framework, a heap graph is represented as a composition of tree components. Sets of heap graphs can then be represented by tuples of tree automata (TA). A fully-automated shape analysis framework based on FA, employing the framework of *abstract regular tree model checking* (ARTMC) [6], has been implemented in the Forester tool [13]. This approach has been shown to handle a wide variety of different dynamically allocated data structures with a performance that compares favourably to other state-of-the-art fully-automated tools.

Our extension of the FA framework allows us to represent relationships between data elements stored inside heap structures. This makes it possible to automatically verify programs that depend on relationships between data, such as various search trees, lists, and skip lists [17], and to also verify, e.g., different sorting algorithms. Technically, we express relationships between data elements associated with nodes of the heap graph by two classes of constraints. *Local data constraints* are associated with transitions of TA and capture relationships between data of neighbouring nodes in a heap graph; they can be used, e.g., to represent ordering internal to some structure such as a binary search tree. *Global data constraints* are associated with states of TA and capture relationships between data in distant parts of the heap. In order to obtain a powerful analysis based on such extended FA, the entire analysis machinery must have been redesigned, including a need to develop mechanisms for propagating data constraints through FA, to adapt the abstraction mechanisms of ARTMC, to develop a new inclusion check between extended FAs, and to define extended abstract transformers.

Our verification method analyzes sequential, non-recursive C programs, and automatically discovers memory safety errors, such as invalid dereferences or memory leaks, and provides an over-approximation of the set of reachable program configurations. Functional properties, such as sortedness, can be checked by adding code that checks pre- and post-conditions. Functional properties can also be checked by querying the computed over-approximation of the set of reachable configurations.

We have implemented our approach as an extension of the Forester tool, which is a gcc plug-in analyzing the intermediate representation generated from C programs. We have applied the tool to verification of data properties, notably sortedness, of sequential programs with data structures, such as various forms of singly- and doubly-linked lists (DLLs), possibly cyclic or shared, binary search trees (BSTs), and even 2-level and 3-level skip lists. The verified programs include operations like insertion, deletion, or reversal, and also bubble-sort and insert-sort both on SLLs and DLLs. The experiments confirm that our approach is not only fully automated and rather general, but also quite efficient, outperforming many previously known approaches even though they are not of the same level of automation or generality. In the case of skip lists, our analysis is the first fully-automated shape analysis which is able to handle skip lists. Our previous fully-automated shape analysis, which did not handle ordering relations, could also handle skip lists automatically [13], but only after modifying the code in such a way that the preservation of the shape invariant does not depend on ordering relations.

Outline. After a review of related works, in Section 3, we present our way of modeling heap graphs by forests. Then, in Section 4, we propose a representation of sets of heap graphs by forest automata that use constraints to specify relationships between data values. Section 5 contains a description of our analysis procedure, including a procedure for saturating the set of constraints over data values. Section 6 outlines how hierarchically nested forest automata can represent more complex data structures. Section 7 describes our implementation of the proposed ideas as well as the obtained experimental results. Section 8 contains conclusions and future work. The appendix contains proofs of some lemmas as well as the source code of the examples that are verified in our experiments.

2 Related Work

As discussed previously, our approach builds on the fully automated FA-based approach for shape analysis of programs with complex dynamic linked data structures [10,11,13]. We significantly extend this approach by allowing it to track ordering relations between data values stored inside dynamic linked data structures.

For shape analysis, many other formalisms than FA have been used, including, e.g., separation logic and various related graph formalisms [21,16,7,9], other logics [19,14], automata [6], or graph grammars [12]. Compared with FA, these approaches typically handle less general heap structures (often restricted to various classes of lists) [21,9], they are less automated (requiring the user to specify loop invariants or at least inductive definitions of the involved data structures) [16,7,9,12], or less scalable [6].

Verification of properties depending on the ordering of data stored in SLLs was considered in [4], which translates programs with SLLs to counter automata. A subsequent analysis of these automata allows one to prove memory safety, sortedness, and termination for the original programs. The work is, however, strongly limited to SLLs. In this paper, we get inspired by the way that [4] uses for dealing with ordering relations on data, but we significantly redesign it to be able to track not only ordering between simple list segments but rather general heap shapes described by FA. In order to achieve this, we had to not only propose a suitable way of combining ordering relations with FA, but we also had to significantly modify many of the operations used over FA.

In [1], another approach for verifying data-dependent properties of programs with lists was proposed. However, even this approach is strongly limited to SLLs, and it is also much less efficient than our current approach. In [2], concurrent programs operating on SLLs are analyzed using an adaptation of a transitive closure logic [3], which also tracks simple sortedness properties between data elements.

Verification of properties of programs depending on the data stored in dynamic linked data structures was considered in the context of the TVLA tool [15] as well. Unlike our approach, [15] assumes a fixed set of shape predicates and uses inductive logic programming to learn predicates needed for tracking non-pointer data. The experiments presented in [15] involve verification of sorting and stability properties of several programs on SLLs (merging, reversal, bubble-sort, insert-sort) as well as insertion and deletion in BSTs. We do not handle stability, but for the other properties, our approach is much faster. Moreover, for BSTs, we verify that a node is greater/smaller than all the nodes in its left/right subtrees (not just than the immediate successors as in [15]).

An approach based on separation logic extended with constraints on the data stored inside dynamic linked data structures and capable of handling size, ordering, as well as bag properties was presented in [8]. Using the approach, various programs with SLLs, DLLs, and also AVL trees and red-black trees were verified. The approach, however, requires the user to manually provide inductive shape predicates as well as loop invariants. Later, the need to provide loop invariants was avoided in [18], but a need to manually provide inductive shape predicates remains.

Another work that targets verification of programs with dynamic linked data structures, including properties depending on the data stored in them, is [22]. It generates verification conditions in an undecidable fragment of higher-order logic and discharges them using decision procedures, first-order theorem proving, and interactive theorem proving. To generate the verification conditions, loop invariants are needed. These can either be provided manually or sometimes synthesized semi-automatically using the approach of [20]. The latter approach was successfully applied to several programs with SLLs, DLLs, trees, trees with parent pointers, and 2-level skip lists. However, for some of them, the user still had to provide some of the needed abstraction predicates.

Several works, including [5], define frameworks for reasoning about pre- and post-conditions of programs with SLLs and data. Decidable fragments, which can express more complex properties on data than we consider, are identified, but the approach does not perform fully automated verification, only checking of pre-post condition pairs.

3 Programs, Graphs, and Forests

We consider sequential non-recursive C programs, operating on a set of variables and the heap, using standard commands and control flow constructs. Variables are either *data variables* or *pointer variables*. Heap cells contain zero or several selector fields and a data field (our framework and implementation extends easily to several data fields). Atomic commands include tests between data variables or fields of heap cells, as well as assignments between data variables, pointer variables, or fields of heap cells. We also support commands for allocation and deallocation of dynamically allocated memory.

Fig. 1 shows an example of a C function inserting a new node into a BST (recall that in BSTs, the data value in a node is larger than all the values of its left subtree and smaller than all the values of its right subtree). Variable `x` descends the BST to find the position at which the node `newNode` with a new data value `d` should be inserted.

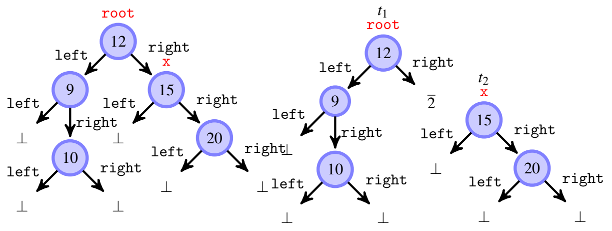
```
Node *insert(Node *root, Data d){
    Node* newNode = calloc(sizeof(Node));
    if (!newNode) return NULL;
    newNode→data = d;
    if (!root) return newNode;
    Node *x = root;
    while (x→data != newNode→data)
        if (x→data < newNode→data)
            if (x→right) x = x→right;
            else x→right = newNode;
        else
            if (x→left) x = x→left;
            else x→left = newNode;
    if (x != newNode) free(newNode);
    return root;
}
```

Fig. 1: A function which inserts a new node into a BST and returns a pointer to its root node

Configurations of the considered programs consist to a large extent of heap-allocated data. A *heap* can be viewed as a (directed) graph whose nodes correspond to allocated memory cells. Each node contains a set of selectors and a data field. Each selector either points to another node, to the value `null`, or is undefined. The same holds for pointer variables of the program.

We represent graphs as a composition of trees as follows. We first identify the *cut-points* of the graph, i.e., nodes that are either referenced by a pointer variable or by several selectors. We then split the graph into tree components such that each cut-point becomes the root of a tree component. To represent the interconnection of tree components, we introduce a set of *root references*, one for each tree component. After decomposition of the graph, selector fields that point to cut-points in the graph are redirected to point to the corresponding root references. Such a tuple of tree components is called a *forest*. The decomposition of a graph into tree components can be performed canonically as described at the end of Section 4.

Fig. 2(a) shows a possible heap of the program in Fig. 1. Nodes are shown as circles, labeled by their data values. Selectors are shown as edges. Each selector points either to a node or to $\perp$ (denoting `null`). Some nodes are labeled by a pointer variable that points to them. The node with data value 15 is a cut-point since it is referenced by variable `x`. Fig. 2(b) shows a tree decomposition of the graph into two trees, one rooted at the node referenced by `root`, and the other rooted at the node pointed by `x`. The `right` selector of the root node in the first tree points to root reference $\overline{2}$ ($\overline{i}$ denotes a reference to the i-th tree t_i) to indicate that in the graph, it points to the corresponding cut-point.



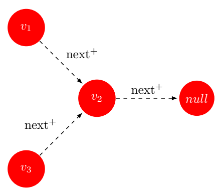
(a) Graph. (b) Forest decomposition.
Fig. 2: Decomposition of a graph into trees.

Let us now formalize these ideas. We will define graphs as parameterized by a set Γ of selectors and a set Ω of references. Intuitively, the references are the objects that selectors can point to, in addition to other nodes. E.g., when representing heaps, Ω will contain the special value `null`; in tree components, Ω will also include root references.

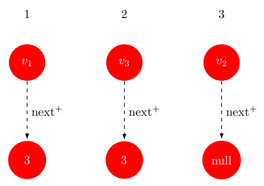
We use $f : A \rightharpoonup B$ to denote a partial function from A to B (also viewed as a total function $f : A \to (B \cup \{\perp\})$, assuming that $\perp \notin B$). We assume an unbounded data domain $\mathbb{D}$ with a total ordering relation $\preceq$.

Graphs. Let Γ be a finite set of *selectors* and Ω be a finite set of *references*. A *graph* g over $\langle \Gamma, \Omega \rangle$ is a tuple $\langle V_g, next_g, \lambda_g \rangle$ where V_g is a finite set of *nodes* (assuming $V_g \cap \Omega = \emptyset$), $next_g : \Gamma \to (V_g \rightharpoonup (V_g \cup \Omega))$ maps each selector $a \in \Gamma$ to a partial mapping $next_g(a)$ from nodes to nodes and references, and $\lambda_g : (V_g \cup \Omega) \rightharpoonup \mathbb{D}$ is a partial *data labelling* of nodes and references. For a selector $a \in \Gamma$, we use a_g to denote the mapping $next_g(a)$.

Program semantics. A *heap* over Γ is a graph over $\langle \Gamma, \{\texttt{null}\} \rangle$ where `null` denotes the null value. A *configuration* of a program with selectors Γ consists of a program control location, a heap g over Γ, and a partial valuation, which maps pointer variables to $V_g \cup \{\texttt{null}\}$ and data variables to $\mathbb{D}$. For uniformity, data variables will be represented as pointer variables (pointing to nodes that hold the respective data values) so we can further consider pointer variables only. The dynamic behaviour of a program is given by a standard mapping from configurations to their successors, which we omit here.



A graph.



A forest representation of the graph shown on the left.

Fig. 3: An example of a forest representation.

Forest representation of graphs. A graph t is a *tree* if its nodes and selectors (i.e., not references) form a tree with a unique root node, denoted $root(t)$. A *forest* over $\langle \Gamma, \Omega \rangle$ is a sequence $t_1 \cdots t_n$ of trees over $\langle \Gamma, (\Omega \uplus \{\overline{1}, \ldots, \overline{n}\}) \rangle$. The element in $\{\overline{1}, \ldots, \overline{n}\}$ are called *root references* (note that n must be the number of trees in the forest). A forest $t_1 \cdots t_n$ is *composable* if $\lambda_{t_k}(\overline{j}) = \lambda_{t_j}(root(t_j))$ for any k, j, i.e., the data labeling of root references agrees with that of roots. A composable forest $t_1 \cdots t_n$ over $\langle \Gamma, \Omega \rangle$ represents a graph over $\langle \Gamma, \{\texttt{null}\} \rangle$, denoted $\otimes t_1 \cdots t_n$, obtained by taking the union of the trees of $t_1 \cdots t_n$ (assuming w.l.o.g. that the sets of nodes of the trees are disjoint), and connecting root references with the corresponding roots. Formally, $\otimes t_1 \cdots t_n$ is the graph g defined by (i) $V_g = \cup_{i=1}^{n} V_{t_i}$, and (ii) for $a \in \Gamma$ and $v \in V_{t_k}$, if $a_{t_k}(v) \in \{\overline{1}, \ldots, \overline{n}\}$ then $a_g(v) = root(t_{a_{t_k}(v)})$ else $a_g(v) = a_{t_k}(v)$, and finally (iii) $\lambda_g(v) = \lambda_{t_k}(v)$ for $v \in V_{t_k}$.

Example 1. Fig. 3 gives an example of how a graph is represented by a forest.

4 Forest Automata

A forest automaton is essentially a tuple of tree automata accepting a set of tuples of trees that represents a set of graphs via their forest decomposition.

Tree automata. A (finite, non-deterministic, top-down) *tree automaton* (TA) over $\langle \Gamma, \Omega \rangle$ extended with data constraints is a triple $A = (Q, q_0, \Delta)$ where Q is a finite set of *states*, $q_0 \in Q$ is the *root state* (or initial state), denoted $root(A)$, and Δ is a set of *transitions*. Each transition is of the form $q \to \overline{a}(q_1, \ldots, q_m) : c$ where $m \geq 0$, $q \in Q$, $q_1, \ldots, q_m \in (Q \cup \Omega)$, $\overline{a} = a^1 \cdots a^m$ is a sequence of different symbols from Γ, and c is a set of *local constraints*. Each local constraint is of the form $0 \sim_{rx} i$ where $\sim \in \{\prec, \preceq, \succ, \succeq\}$ (with $=$ and $\neq$ viewed as syntactic sugar), $i \in \{1, \ldots, m\}$, and $x \in \{r, a\}$. Intuitively, a local constraint of the form $0 \sim_{rr} i$ states that the data value of the *root* of every tree t accepted at q is related by $\sim$ with the data value of the *root* of the ith subtree of t accepted at q_i. A local constraint of the form $0 \sim_{ra} i$ states that the data value of the *root* of every tree t accepted at q is related by $\sim$ to the data values of *all* nodes of the i-th subtree of t accepted at q_i.

Let t be a tree over $\langle \Gamma, \Omega \rangle$, and let $A = (Q, q_0, \Delta)$ be a TA over $\langle \Gamma, \Omega \rangle$. A *run* of A over t is a total map $\rho : V_t \to Q$ where $\rho(root(t)) = q_0$ and for each node $v \in V_t$ there is a transition $q \to \overline{a}(q_1, \ldots, q_m) : c$ in Δ with $\overline{a} = a^1 \cdots a^m$ such that (1) $\rho(v) = q$, (2) for all $1 \le i \le m$, we have (i) if $q_i \in Q$, then $a^i_t(v) \in V_t$ and $\rho(a^i_t(v)) = q_i$, and (ii) if $q_i \in \Omega$, then $a^i_t(v) = q_i$, and (3) for each constraint in c, the following holds:

- if the constraint is of the form $0 \sim_{rr} i$, then $\lambda_t(v) \sim \lambda_t(a^i_t(v))$, and
- if the constraint is of the form $0 \sim_{ra} i$, then $\lambda_t(v) \sim \lambda_t(w)$ for all nodes w in V_t that are in the subtree of t rooted at $a^i_t(v)$.

We define the *language* of A as $L(A) = \{t \mid \text{there is a run of } A \text{ over } t\}$.

Example 2. BSTs, like the tree labeled by `x` in Fig. 2, are accepted by the TA with one state q_1, which is also the root state, and the following four transitions:

$$\begin{array}{llll} q_1 \to \texttt{left},\texttt{right}(q_1,q_1) & : 0 \succ_{ra} 1, 0 \prec_{ra} 2 & q_1 \to \texttt{left},\texttt{right}(q_1,\texttt{null}) & : 0 \succ_{ra} 1 \\ q_1 \to \texttt{left},\texttt{right}(\texttt{null},q_1) & : 0 \prec_{ra} 2 & q_1 \to \texttt{left},\texttt{right}(\texttt{null},\texttt{null}) & \end{array}$$

The local constraints of the transitions express that the data value in a node is always greater than the data values of all nodes in its left subtree and less than the data values of all nodes in its right subtree.

A TA that accepts BSTs in which the `right` selector of the root node points to a root reference, like that labeled by `root` in Fig. 2, can be obtained from the above TA by adding one more state q_0, which then becomes the root state, and the additional transition $q_0 \to \texttt{left},\texttt{right}(q_1,\overline{2}) : 0 \succ_{ra} 1, 0 \prec_{rr} 2$ (note that the occurrence of 2 in the root reference $\overline{2}$ is not related with the occurrence of 2 in the local constraint). □

Forest automata. A *forest automaton with data constraints* (or simply a forest automaton, FA) over $\langle \Gamma, \Omega \rangle$ is a tuple of the form $F = \langle A_1 \cdots A_n, \varphi \rangle$ where:

- $A_1 \cdots A_n$, with $n \ge 0$, is a sequence of TA over $\langle \Gamma, \Omega \uplus \{\overline{1}, \ldots, \overline{n}\} \rangle$ whose sets of states $Q_1, \ldots, Q_n$ are mutually disjoint.
- φ is a set of *global data constraints* between the states of $A_1 \cdots A_n$, each of the form $q \sim_{rr} q'$ or $q \sim_{ra} q'$ where $q, q' \in \cup_{i=1}^{n} Q_i$, at least one of q, q' is a root state which does not appear on the right-hand side of any transition (i.e., it can accept only the root of a tree), and $\sim \in \{\prec, \preceq, \succ, \succeq\}$ (with $=$ and $\neq$ viewed as syntactic sugar). Intuitively, $q \sim_{rr} q'$ says that the data value of any tree node accepted at q is related by $\sim$ to the data value of any tree node accepted at q'. Similarly, $q \sim_{ra} q'$ says that the data value of any tree node accepted at q is related by $\sim$ to the data values of *all* nodes of the trees accepted at q'.

A forest $t_1 \cdots t_n$ over $\langle \Gamma, \Omega \rangle$ is *accepted* by F iff there are runs $\rho_1, \ldots, \rho_n$ such that ρ_i is a run of A_i over t_i for every $1 \le i \le n$, and for each global constraint of the form $q \sim_{rx} q'$ where q is a state of some A_i and q' is a state of some A_j, we have

- if $rx = rr$, then $\lambda_{t_i}(v) \sim \lambda_{t_j}(v')$ whenever $\rho_i(v) = q$ and $\rho_j(v') = q'$,
- if $rx = ra$, then $\lambda_{t_i}(v) \sim \lambda_{t_j}(w)$ whenever $\rho_i(v) = q$ and w is in a subtree rooted at some v' with $\rho_j(v') = q'$.

The *language* of F, denoted as $L(F)$, is the set of graphs over $\langle \Gamma, \Omega \rangle$ obtained by applying $\otimes$ on composable forests accepted by F. An FA F over $\langle \Gamma, \{\texttt{null}\} \rangle$ represents a set of heaps H over Γ.

Note that global constraints can imply some local ones, but they cannot in general be replaced by local constraints only. Indeed, global constraints can relate states of different automata as well as states that do not appear in a single transition and hence accept nodes which can be arbitrarily far from each other and unrelated by any sequence of local constraints.

Canonicity. In our analysis, we will represent only *garbage-free* heaps in which all nodes are reachable from some pointer variable by following some sequence of selectors. In practice, this is not a restriction since emergence of garbage is checked for each statement in our analysis; if some garbage arises, an error message can be issued, or the garbage removed. The representation of a garbage-free heap H as $t_1 \cdots t_n$ can be made canonical by assuming a total order on variables and on selectors. Such an ordering induces a canonical ordering of cut-points using a depth-first traversal of H starting from pointer variables, taken in their order, and exploring H according to the order of selectors. The representation of H as $t_1 \cdots t_n$ is called *canonical* iff the roots of the trees in $t_1 \cdots t_n$ are the cut-points of H, and the trees are ordered according to their canonical ordering. An FA $F = \langle A_1 \cdots A_n, \varphi \rangle$ is *canonicity respecting* iff for all $H \in L(F)$, formed as $H = \otimes t_1 \cdots t_n$, the representation $t_1 \cdots t_n$ is canonical. The canonicity respecting form allows us to check inclusion on the sets of heaps represented by FA by checking inclusion component-wise on the languages of the component TA.

5 FA-based Shape Analysis with Data

Our verification procedure performs a standard abstract interpretation. The concrete domain in our case assigns to each program location a set of pairs $\langle \sigma, H \rangle$ where the *valuation* σ maps every variable to `null`, a node in H, or to an undefined value, and H is a heap representing a memory configuration. On the other hand, the abstract domain maps each program location to a finite set of *abstract configurations*. Each abstract configuration is a pair $\langle \sigma, F \rangle$ where σ maps every variable to `null`, an index of a TA in F, or to an undefined value, and F is an FA representing a set of heaps.

Example 3. The example illustrates an abstract configuration $\langle \sigma, F \rangle$ encoding a single concrete configuration $\langle \sigma, H \rangle$ of the program in Fig. 1. A memory node referenced by `newNode` is going to be added as the left child of the leaf referenced by `x`, which is reachable from the root by the sequence of selectors `left right`. The data values along the path from `root` to `x` must be in the proper relations with the data value of `newNode`, in order for the tree to stay sorted also after the addition. The data value of `newNode` must be smaller than that of the root (i.e., $q_{\mathtt{r}} \succ_{ra} q_{\mathtt{nN}}$), larger than that of its left child (i.e., $q \prec_{ra} q_{\mathtt{nN}}$), and smaller than that of `x` (i.e., $q_{\mathtt{x}} \succ_{ra} q_{\mathtt{nN}}$). These relations and also $q \prec_{ra} q_{\mathtt{x}}$ have been accumulated during the tree traversal. □

$$
\begin{array}{l}
F = \langle A_1 A_2 A_3, \varphi \rangle \\
\sigma(\mathtt{root}) = \overline{1}, \sigma(\mathtt{x}) = \overline{2}, \sigma(\mathtt{newNode}) = \overline{3} \\
A_1 : \begin{cases} q_{\mathtt{r}} \to \mathtt{left}, \mathtt{right}(q, \mathtt{null}) : 0 \succ_{ra} 1 \\ q \to \mathtt{left}, \mathtt{right}(\mathtt{null}, \overline{2}) : \ 0 \prec_{ra} 2 \end{cases} \\
A_2 : q_{\mathtt{x}} \to \mathtt{left}, \mathtt{right}(\mathtt{null}, \mathtt{null}) \\
A_3 : q_{\mathtt{nN}} \to \mathtt{left}, \mathtt{right}(\mathtt{null}, \mathtt{null}) \\
\varphi = \{ q_{\mathtt{r}} \succ_{ra} q_{\mathtt{nN}}, q \prec_{ra} q_{\mathtt{nN}}, q_{\mathtt{x}} \succ_{ra} q_{\mathtt{nN}}, q \prec_{ra} q_{\mathtt{x}} \}
\end{array}
$$

The verification starts from an element in the abstract domain that represents the initial program configuration (i.e., it maps the initial program location to an abstract configuration where the heap is empty and the values of all variables are undefined, and maps non-initial program locations to an empty set of abstract configurations). The verification then iteratively updates the sets of abstract configurations at each program point until a fixpoint is reached. Each iteration consists of the following steps:

1. The sets of abstract configurations at each program point are updated by abstract transformers corresponding to program statements. At junctions of program paths, we take the unions of the sets produced by the abstract transformers.
2. At junctions that correspond to loop points, the union is followed by a widening operation and a check for language inclusion between sets of FA in order to determine whether a fixpoint has been reached. Prior to checking language inclusion, we normalize the FA, thereby transforming them into the canonicity respecting form, which is needed for inclusion checking as explained at the end of Section 4.

Our widening operation bounds the size of the TA that occur in abstract configurations. It is based on the framework of *abstract regular (tree) model checking* [6]. The widening is applied to individual TA inside each FA and collapses states which are equivalent w.r.t. certain criteria. More precisely, we collapse TA states q, q' which are equivalent in the sense that they (1) accept trees with the same sets of prefixes of height at most k and (2) occur in isomorphic global data constraints (i.e., $q \sim_{rx} p$ occurs as a global constraint if and only if $q' \sim_{rx} p$ occurs as a global constraint, for any p and x). We use a refinement of this criterion by certain FA-specific requirements, by adapting the refinement described in [13]. Collapsing states may increase the set of trees accepted by a TA, thereby introducing overapproximation into our analysis.

At the beginning of each iteration, the FA to be manipulated are in the saturated form, meaning that they explicitly include all (local and global) data constraints that are consequences of the existing ones. FA can be put into a saturated form by a saturation procedure, which is performed before the normalization procedure. The saturation procedure must also be performed before applying abstract transformers that may remove root states from an FA, such as memory deallocation.

In the following subsections, we provide more detail on some of the major steps of our analysis. Section 5.1 describes the constraint saturation procedure, Section 5.2 describes some representative abstract transformers, Section 5.3 describes normalization, and Section 5.4 describes our check for inclusion.

5.1 Constraint Saturation

In the analysis, we work with FA that are saturated by explicitly adding into them various (local and global) data constraints that are implied by the existing ones. The saturation is based on applying several saturation rules, each of which infers new constraints from the existing ones, until no more rules can be applied. Because of space limitations, we present here only a representative sample of the rules. A complete description of our saturation rules can be found in Appendix A. Our saturation rules can be structured into the following classes.

– New global constraints can be inferred from existing global constraints by using transitivity, reflexivity, and symmetry (when applicable) of the involved relations. For instance, from $q \preceq_{rr} q'$ and $q' \prec_{ra} q''$, we infer $q \prec_{ra} q''$ by transitivity.
– New global or local constraints can be inferred by weakening the existing ones. For instance, from $q \prec_{ra} q'$, we infer the weaker constraint $q \preceq_{rr} q'$.
– Each local constraint $0 \prec_{rr} i$ where $q_i \in \Omega$ or q_i has nullary outgoing transitions only can be strengthened to $0 \prec_{ra} i$. The latter applies to global transitions too.
– New local constraints can be inferred from global ones by simply transforming a global constraint into a local constraint whenever the states in a transition are related by a global constraint. For instance, if $q \to \overline{a}(q_1, \ldots, q_m) : c$ is a transition, then from $q \preceq_{rr} q_i$, we infer the local constraint $0 \preceq_{rr} i$ and add it to c.
– If q is a state of a TA A and p is a state of A or another TA of the given FA such that in each sequence of states through which q can be reached from the root state of A there is a state q' such that $p \sim_{ra} q'$, then a constraint $p \sim_{ra} q$ is added as well.
– Whenever there is a TA A_1 with a root state q_0 and a state q such that (i) $q_0 \succeq_{rr} q$, (ii) q has an outgoing transition in whose right-hand side a state q_i appears where q_i is a reference to a TA A_2, and (iii) c includes a constraint $0 \succeq_{rr} i$, then a global constraint $q_0 \succeq_{rr} p_0$ can be added for the root state p_0 of A_2 (likewise for other kinds of relations than $\succeq_{rr}$). Conversely, from $q_0 \succeq_{rr} p_0$ and $q_0 \succeq_{rr} q$, one can derive the local constrain $0 \succeq_{rr} i$.
– Finally, global constraints can be inferred from existing ones by propagating them over local constraints of transitions in which the states of the global constraints occur. Let us illustrate this on a small example. Assume we are given a TA A that has states $\{q_0, q_1, q_2\}$ with q_0 being the root state and the following transitions: $q_0 \to \overline{a}(q_1, q_2) : \{0 \prec_{rr} 1, 0 \prec_{rr} 2\}$, $q_1 \to \overline{a}(\texttt{null}, \texttt{null}) : \emptyset$, and $q_2 \to \overline{a}(\texttt{null}, \texttt{null}) : \emptyset$. Let p be a root state of some TA in an FA in which A appears. There are two ways to propagate global constraints between the states of A, either *downwards* from the root towards leaves or *upwards* from leaves towards the root.
  - In downwards propagation, we can infer $q_2 \succ_{ra} p$ from $q_0 \succeq_{ra} p$, using the local constraint $0 \prec_{rr} 2$.
  - In upwards propagation, we can infer $q_0 \prec_{rr} p$ from $q_2 \prec_{rr} p$, using the local constraint $0 \prec_{rr} 2$.

  In more complex situations, a single state may be reached in several different ways. In such cases, propagation of global constraints through local constraints on all transitions arriving to the given state must be considered. If some of the ways how to get to the state does not allow the propagation, it cannot be done. Moreover, since one propagation can enable another one, the propagation must be done iteratively until a fixpoint is reached (for more details, see Appendix A). Note that the iterative propagation must terminate since the number of constraints that can be used is finite.

5.2 Abstract Transformers

For each operation op in the intermediate representation of the analysed program corresponding to the function $f_{\texttt{op}}$ on concrete configurations $\langle \sigma, H \rangle$, we define an abstract transformer $\tau_{\texttt{op}}$ on abstract configurations $\langle\sigma, F\rangle$ such that the result of $\tau_{\texttt{op}}(\langle\sigma, F\rangle)$ denotes the set $\{f_{\texttt{op}}(\langle\sigma, H\rangle) \mid H \in L(F)\}$. The abstract transformer $\tau_{\texttt{op}}$ is applied separately for each pair $\langle\sigma, F\rangle$ in an abstract configuration. Note that all our abstract transformers $\tau_{\texttt{op}}$ are exact.

Let us present the abstract transformers corresponding to some operations on abstract states of form $\langle\sigma, F\rangle$. For simplicity of presentation, we assume that for all TA A_i in F, (a) the root state of A_i does not appear in the right-hand side of any transition, and (b) it occurs on the left-hand side of exactly one transition. It is easy to see that any TA can be transformed into this form (see Appendix **??** for details).

Let us introduce some common notation and operations for the below transformers. We use $A_{\sigma(\texttt{x})}$ and $A_{\sigma(\texttt{y})}$ to denote the TA pointed by variables x and y, respectively, and $q_{\texttt{x}}$ and $q_{\texttt{y}}$ to denote the root states of these TA. Let $q_{\texttt{y}} \to \overline{a}(q_1, \ldots, q_i, \ldots, q_m) : c$ be the unique transition from $q_{\texttt{y}}$. We assume that `sel` is represented by a^i in the sequence $\overline{a} = a^1 \cdots a^m$ so that q_i corresponds to the target of `sel`. By *splitting* a TA $A_{\sigma(\texttt{y})}$ at a state q_i for $1 \leq i \leq m$, we mean appending a new TA A_k to F such that A_k is a copy of $A_{\sigma(\texttt{y})}$ but with q_i as the root state, followed by changing the root transition in $A_{\sigma(\texttt{y})}$ to $q_{\texttt{y}} \to \overline{a}(q_1, \ldots, \overline{k}, \ldots, q_m) : c'$ where c' is obtained from c by replacing any local constraint of the form $0 \sim_{rx} i$ by the global constraint $q_{\texttt{y}} \sim_{rx} root(A_k)$. Global data constraints are adapted as follows: For each constraint $q \sim_{rx} p$ where q is in $A_{\sigma(\texttt{y})}$ such that $q \neq q_y$, a new constraint $q' \sim_{rx} p$ is added. Likewise, for each constraint $q \sim_{rx} p$ where p is in $A_{\sigma(\texttt{y})}$ such that $p \neq q_y$, a new constraint $q \sim_{rx} p'$ is added. Finally, for each constraint of the form $p \sim_{ra} q_{\texttt{y}}$, a new constraint $p \sim_{ra} root(A_k)$ is added.

Before performing the actual update, we check whether the operation to be performed tries to dereference a pointer to `null` or to an undefined value, in which case we stop the analysis and report an error. Otherwise, we continue by performing one of the following actions, depending on the particular statement:

x = `malloc()` We extend F with a new TA A_{new} containing one state and one transition where all selector values are undefined and assign $\sigma(\texttt{x})$ to the index of A_{new} in F.

x = **y->sel** If q_i is a root reference (say, j), it is sufficient to change the value of $\sigma(\texttt{x})$ to j. Otherwise, we split $A_{\sigma(\texttt{y})}$ at q_i (creating A_k) and assign k to $\sigma(\texttt{x})$.

y->sel = x If q_i is a state, then we split $A_{\sigma(\texttt{y})}$ at q_i. Then we put $\overline{\sigma(\texttt{x})}$ to the i-th position in the right-hand side of the root transition of $A_{\sigma(\texttt{y})}$; this is done both if q_i is a state and if q_i is a root reference. Any local constraint in c of the form $0 \sim_{rx} i$ which concerns the removed root reference q_i is then removed from c.

y->data = **x->data** First, we remove any local constraint that involves $q_{\texttt{y}}$ or a root reference to $A_{\sigma(\texttt{y})}$. Then, we add a new global constraint $q_{\texttt{y}} =_{rr} q_{\texttt{x}}$, and we also keep all global constraints of the form $q' \sim_{rx} q_{\texttt{y}}$ if $q' \sim_{rr} q_{\texttt{x}}$ is implied by the constraints obtained after the update.

y->data $\sim$ **x->data** (where $\sim \in \{\prec, \preceq, \succ, \succeq, =, \neq\}$) First, we execute the saturation procedure in order to infer the strongest constraints between $q_{\texttt{y}}$ and $q_{\texttt{x}}$. Then, if there exists a global constraint $q_{\texttt{y}} \sim' q_{\texttt{x}}$ that implies $q_{\texttt{y}} \sim q_{\texttt{x}}$ (or its negation), we return *true* (or *false*). Otherwise, we copy $\langle\sigma, F\rangle$ into two abstract configurations: $\langle\sigma, F_{true}\rangle$ for the *true* branch and $\langle\sigma, F_{false}\rangle$ for the *false* branch. Moreover, we extend F_{true} with the global constraint $q_{\texttt{y}} \sim q_{\texttt{x}}$ and F_{false} with its negation.

x = y **or** x = `NULL` We simply update σ accordingly.

`free(y)` First, we split $A_{\sigma(\texttt{y})}$ at all states q_j, $1 \leq j \leq m$, that appear in its root transition, then we remove $A_{\sigma(\texttt{y})}$ from F and set $\sigma(\texttt{y})$ to undefined. However, to keep all possible data constraints, before removing $A_{\sigma(\texttt{y})}$, the saturation procedure is executed. After the action is done, every global constraint involving $q_{\texttt{y}}$ is removed.

`x == y` This operation is evaluated simply by checking whether $\sigma(\texttt{x}) = \sigma(\texttt{y})$. If $\sigma(\texttt{x})$ or $\sigma(\texttt{y})$ is undefined, we assume both possibilities.

After the update, we check that all TA in F are referenced, either by a variable or from a root reference, otherwise we report emergence of garbage.

5.3 Normalization

Normalization transforms an FA $F = (A_1 \cdots A_n, \varphi)$ into a canonicity respecting FA in three major steps:

1. First, we transform F into a form in which roots of trees of accepted forests correspond to cut-points in a uniform way. In particular, for all $1 \leq i \leq n$ and all accepted forests $t_1 \cdots t_n$, one of the following holds: (a) If the root of t_i is the j-th cut-point in the canonical ordering of an accepted forest, then it is the j-th cut-point in the canonical ordering of all accepted forests. (b) Otherwise the root of t_i is not a cut-point of any of the accepted forests.
2. Then we merge TA so that the roots of trees of accepted forests are cut-points only, which is described in detail below.
3. Finally, we reorder the TA according to the canonical ordering of cut-points (which are roots of the accepted trees).

Our procedure is an augmentation of that in [10,11] used to normalize FA without data constraints. The difference, which we describe below, is an update of data constraints while performing Step 2.

In order to minimize a possible loss of information encoded by data constraints, Step 2 is preceded by saturation (Section 5.1). Then, for all $1 \leq i \leq n$ such that roots of trees accepted by $A_i = (Q_A, q_A, \Delta_A)$ are not cut-points of the graphs in $L(F)$ and such that there is a TA $B = (Q_B, q_B, \Delta_B)$ that contains a root reference to A_i, Step 2 performs the following. The TA A_i is removed from F, data constraints between q_A and non-root states of F are removed from φ, and A_i is connected to B at the places where B refers to it. In detail, B is replaced by the TA $(Q_A \cup Q_B, q_B, \Delta_{A+B})$ where Δ_{A+B} is constructed from $\Delta_A \cup \Delta_B$ by modifying every transition $q \to \overline{a}(q_1, \ldots, q_m) : c \in \Delta_B$ as follows:

1. all occurrences of $\bar{i}$ among $q_1, \ldots, q_m$ are replaced by q_A, and
2. for all $1 \leq k \leq m$ s.t. q_k can reach $\bar{i}$ by following top-down a sequence of the original rules of Δ_B, the constraint $0 \sim_{ra} k$ is removed from c unless $q_k \sim_{ra} q_A \in \varphi$.

5.4 Checking Language Inclusion

In this section, we describe a reduction of checking language inclusion of FAs with data constraints to checking language inclusion of FAs without data constraints, which can be then performed by the techniques of [10,11]. We note that "ordinary FAs" correspond to FAs with no global and no local data constraints. Intuitively, an *encoding* of an FA $F = (A_1 \cdots A_n, \varphi)$ with data constraints is an ordinary FA $F^E = (A_1^E \cdots A_n^E, \emptyset)$ where the data constraints are written into symbols of transitions. In detail, each transition $q \to \overline{a}(q_1, \ldots, q_m) : c$ of $A_i, 1 \leq i \leq n$, is in A_i^E replaced by the transition $q \to \langle (a_1, c_1, c_g) \cdots (a_m, c_m, c_g) \rangle (q_1, \ldots, q_m) : \emptyset$ where for $1 \leq j \leq m$, c_j is the subset of c involving j, and c_g encodes the global constraints involving q as follows: for a global constraint $q \sim_{rx} r$ or $r \sim_{rx} q$ where r is the root state of $A_k, 1 \leq k \leq n$, that does not appear within any right-hand side of a rule, c_g contains $0 \sim_{rx} k$ or $k \sim_{rx} 0$, respectively. The language of A_i^E thus consists of trees over the alphabet $\Gamma^E = \Gamma \times \mathbb{C} \times \mathbb{C}$ where $\mathbb{C}$ is the set of constraints of the form $j \sim_{rx} k$ for $j, k \in \mathbb{N}_0$.

Dually, a *decoding* of a forest $t_1 \cdots t_n$ over Γ^E is the set of forests $t_1' \cdots t_n'$ over Γ which arise from $t_1 \cdots t_n$ by (1) removing encoded constraints from the symbols, and (2) choosing data labeling that satisfies the constraints encoded within the symbols of $t_1 \cdots t_n$. Formally, for all $1 \leq i \leq n$, $V_{t_i'} = V_{t_i}$, and for all $a \in \Gamma$, $u, v \in V_{t_i'}$, and $c, c_g \subseteq \mathbb{C}$, we have $(a, c, c_g)_{t_i}(u) = v$ iff: (1) $a_{t_i'}(u) = v$ and (2) for all $1 \leq j \leq n$: if $0 \sim_{rx} j \in c$, then $u \sim_{rx} v$, and if $0 \sim_{rx} j \in c_g$, then $u \sim_{rx} \mathit{root}(t_j)$ (symmetrically for $j \sim_{rx} 0$). The notation $u \sim_{rx} v$ for $u, v \in V_{t'}$ used here has the expected meaning that $\lambda_{t_i'}(u) \sim \lambda_{t_i'}(v)$ and, in case of $x = a$, $\lambda_{t_i'}(u) \sim \lambda_{t_i'}(w)$ for all nodes w in the subtree rooted by v.

The following lemma (proved in Appendix C) assures that encodings of FA are related in the expected way with decodings of forests they accept.

Lemma 1. *The set of forests accepted by an FA F is equal to the union of decodings of forests accepted by F^E.*

A direct consequence of Lemma 1 is that if $L(F_A^E) \subseteq L(F_B^E)$, then $L(F_A) \subseteq L(F_B)$. We can thus use the language inclusion checking procedure of [10,11] for ordinary FA to safely approximate language inclusion of FA with data constraints.

However, the above implication of inclusions does not hold in the opposite direction, for two reasons. First, constraints of F_B that are strictly weaker than constraints of F_A will be translated into different labels. The labels will then be treated as *incomparable* by the inclusion checking algorithm of [10,11]. For instance, let $F_A = (A_1, \emptyset)$ where A_1 contains only one transition $\delta_A = q \to a(\overline{1}) : \{0 \prec_{rr} 1\}$ and $F_B = (B_1, \emptyset)$ where B_1 contains only one transition $\delta_B = r \to a(\overline{1}) : \emptyset$. We have that $L(F_A) \subseteq L(F_B)$ (indeed, $L(F_A) = \emptyset$ due to the strict inequality on the root), but $L(F_A^E)$ is incomparable with $L(F_B^E)$. The reason is that δ_A and δ_B are encoded as transitions the symbols of which differ due to different data constraints. The fact that the constraint $\emptyset$ is weaker than the constraint of $0 \prec_{rr} 1$ plays no role. The second source of incompleteness of our inclusion checking procedure is that decodings of some forests accepted by F_A^E and F_B^E may be empty due to inconsistent data constraints. If the set of such inconsistent forests of F_A^E is not included in that of F_B^E, then $L(F_A^E)$ cannot be included in $L(F_B^E)$, but the inclusion $L(F_A) \subseteq L(F_B)$ can still hold since the forests with the empty decodings do not contribute to $L(F_A)$ and $L(F_B)$ (in the sense of Lemma 1).

We do not attempt to resolve the second difficulty since ruling out forests with inconsistent data constraints seems to be complicated, and according to our experiments, it does not seem necessary. On the other hand, we resolve the first difficulty by a quite simple transformation of F_B^E: we pump up the TAs of F_B^E by variants of their transitions which encode stronger data constraints than originals and match the data constraints on transitions of F_A^E. For instance, in our previous example, we wish to add the transition $r \to a(\overline{1}) : \{0 \prec_{rr} 1\}$ to B_1. Notice that this does not change the language of F_B, but makes checking of $L(F_A^E) \subseteq L(F_B^E)$ pass.

Particularly, we call a sequence $\overline{\alpha} = (a_1, c_1, c_g) \cdots (a_m, c_m, c_g) \in (\Gamma^E)^m$ *stronger* than a sequence $\overline{\beta} = (a_1, c'_1, c'_g) \cdots (a_m, c'_m, c'_g)$ iff $\bigwedge c_g \implies \bigwedge c'_g$ and for all $1 \leq i \leq m$, $\bigwedge c_i \implies \bigwedge c'_i$. Intuitively, $\overline{\alpha}$ encodes the same sequence of symbols $\overline{a} = a_1 \cdots a_m$ as $\overline{\beta}$ and stronger local and global data constraints than $\overline{\beta}$. We modify F_B^E in such a way that for each transition $r \to \overline{\alpha}(r_1, \ldots, r_m)$ of F_B^E and each transition of F_A^E of the form $q \to \overline{\beta}(q_1, \ldots, q_m)$ where $\overline{\beta}$ is stronger than $\overline{\alpha}$, we add the transition $q \to \overline{\beta}(q_1, \ldots, q_m)$. The modified FA, denoted by $F_B^{E^+}$, accepts the same or more forests than F_B^E (since its TA have more transitions), but the sets of decodings of the accepted forests are the same (since the added transitions encode stronger constraints than the existing transitions). FA $F_B^{E^+}$ can thus be used within language inclusion checking in the place of F_B^E. The checking is still sound, and the chance of missing inclusion is smaller. The following lemma (proved in Appendix C) summarises soundness of the (approximation of) inclusion check which is implemented in our tool.

Lemma 2. *Given two FAs F_A and F_B, $L(F_A^E) \subseteq L(F_B^{E^+}) \implies L(F_A) \subseteq L(F_B)$*

We note that the same construction is used when checking language inclusion between sets of FAs with data constraints in a combination with the construction of [10,11] for checking inclusion of sets of ordinary FAs. We also note that for the purpose of checking language inclusion, we need to work with TAs where the tuples $\overline{a}$ of symbols (selectors) on all rules are ordered according to a fixed total ordering of selectors (we use the one from Section 4, used to define canonical forests).

6 Boxes

Forest automata, as defined in Section 4, cannot be used to represent sets of graphs with an unbounded number of cut-points since this would require an unbounded number of TAs within FAs. An example of such a set of graphs is the set of all DLLs of an arbitrary length where each internal node is a cut-point. The solution provided in [10,11] is to allow FAs to use other nested FAs, called boxes, as symbols to "hide" recurring subgraphs and in this way eliminate cut-points. Here, we give only an informal description of a simplified version of boxes from [10,11] and of their combination with data constraints. See Appendix B for details.

A *box* $\square = \langle F_\square, i, o \rangle$ consists of an FA $F_\square = \langle A_1 \cdots A_n, \varphi \rangle$ accompanied with an *input port index i* and an *output port index o*, $1 \leq i, o \leq n$. Boxes can be used as symbols in the alphabet of another FA F. A graph g from $L(F)$ over an alphabet Γ enriched with boxes then represents a set of graphs over Γ obtained by the operation of *unfolding*. Unfolding replaces an edge with a box label $\square$ by a graph $g_\square \in L(F_\square)$. The node of $g_\square$ which is the root of a tree accepted by A_i is identified with the source of the replaced edge, and the node of $g_\square$ which is the root of a tree accepted by A_o is mapped to the

Table 1: Results of the experiments

Example	time	Example	time	Example	time	Example	time
SLL insert	0.06	DLL insert	0.14	BST insert	6.87	SL_2 insert	9.65
SLL delete	0.08	DLL delete	0.38	BST delete	114.00	SL_2 delete	10.14
SLL reverse	0.07	DLL reverse	0.16	BST left rotate	7.35	SL_3 insert	56.99
SLL bubblesort	0.13	DLL bubblesort	0.39	BST right rotate	6.25	SL_3 delete	57.35
SLL insertsort	0.10	DLL insertsort	0.43				

target of the edge. The *semantics* of F then consists of all fully unfolded graphs from the language of F. The alphabet of a box itself may also include boxes, however, these boxes are required to form a hierarchy, they cannot be recursively nested.

In a verification run, boxes are automatically inferred using the techniques presented in [13]. Abstraction is combined with *folding*, which substitutes substructures of FAs by TA transitions which use boxes as labels. On the other hand, *unfolding* is required by abstract transformers that refer to nodes or selectors encoded within a box to expose the content of the box by making it a part of the top-level FA.

In order not to loose information stored within data constraints, folding and unfolding require some additional calls of the saturation procedure. When folding, saturation is used to transform global constraints into local ones. Namely, global constraints between the root state of the TA which is to become the input port of a box and the state of the TA which is to become the output port of the box is transformed into a local constraint of the newly introduced transition which uses the box as a label. When unfolding, saturation is used to transform local constraints into global ones. Namely, local constraints between the left-hand side of the transition with the unfolded box and the right-hand side position attached to the unfolded box is transformed to a global constraint between the root states of the TA within the box which correspond to its input and output port.

7 Experimental Results

We have implemented the above presented techniques as an extension of the Forester tool and tested their generality and efficiency on a number of case studies. We considered programs dealing with SLLs, DLLs, BSTs, and skip lists. We verified the original implementation of skip lists that uses the data ordering relation to detect the end of the operated window (as opposed to the implementation handled in [13] which was modified to remove the dependency of the algorithm on sortedness).

Table 1 gives running times in seconds (the average of 10 executions) of the extension of Forester on our case studies. The names of the examples in the table contain the name of the data structure manipulated in the program, which is "SLL" for singly-linked lists, "DLL" for doubly-linked lists, and "BST" for binary search trees. "SL" stands for skip lists where the subscript denotes their level (the total number of `next` pointers in each cell). All experiments start with a random creation of an instance of the specified structure and end with its disposal. The indicated procedure is performed in between. The "insert" procedure inserts a node into an ordered instance of the structure, at the position given by the data value of the node, "delete" removes the first node with a particular data value, and "reverse" reverses the structure. "Bubblesort" and "insertsort" perform the given sorting algorithm on an unordered instance of the list. "Left rotate" and "right rotate" rotate the BST in the specified direction. Before the disposal of the data structure, we further check that it remained ordered after execution of the operation. Source code of the case studies can be found in Appendix D. The experiments were run on a machine with the Intel i5 M 480 (2.67 GHz) CPU and 5 GB of RAM.

Compared with works [15,20,4,18], which we consider the closest to our approach, the running times show that our approach is significantly faster. We, however, note that a precise comparison is not easy even with the mentioned works since as discussed in the related work paragraph, they can handle more complex properties on data, but on the other hand, they are less automated or handle less general classes of pointer structures.

8 Conclusion

We have extended the FA-based analysis of heap manipulating programs with a support for reasoning about data stored in dynamic memory. The resulting method allows for verification of pointer programs where the needed inductive invariants combine complex shape properties with constraints over stored data, such as sortedness. The method is fully automatic, quite general, and its efficiency is comparable with other state-of-the-art analyses even though they handle less general classes of programs and/or are less automated. We presented experimental results from verifying programs dealing with variants of (ordered) lists and trees. To the best of our knowledge, our method is the first one to cope fully automatically with a full C implementation of a 3-level skip list.

We conjecture that our method generalises to handle other types of properties in the data domain (e.g., comparing sets of stored values) or other types of constraints (e.g., constraints over lengths of lists or branches in a tree needed to express, e.g., balancedness of a tree). We are currently working on an extension of FA that can express more general classes of shapes (e.g., B+ trees) by allowing recursive nesting of boxes, and employing the CEGAR loop of ARTMC. We also plan to combine the method with techniques to handle concurrency.

Acknowledgement. This work was supported by the Czech Science Foundation (projects P103/10/0306, 13-37876P), the Czech Ministry of Education, Youth, and Sports (project MSM 0021630528), the BUT FIT project FIT-S-12-1, the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070, the Swedish Foundation for Strategic Research within the ProFuN project, and by the Swedish Research Council within the UPMARC centre of excellence.

References

1. P. Abdulla, M. Atto, J. Cederberg, and R. Ji. Automated Analysis of Data-Dependent Programs with Dynamic Memory. In *Proc. of ATVA'09*, *LNCS* 5799. Springer, 2009.
2. P. Abdulla, F. Haziza, L. Holík, B. Jonsson, and A. Rezine. An Integrated Specification and Verification Technique for Highly Concurrent Data Structures. *Proc of TACAS'13*, *LNCS* 7795. Springer, 2013.

3. J. Bingham and Z. Rakamaric. A Logic and Decision Procedure for Predicate Abstraction of Heap-Manipulating Programs. In *Proc. of VMCAI'06*, *LNCS* 3855. Springer, 2006.
4. A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with Lists Are Counter Automata. *Formal Methods in System Design*, 38(2):158–192, 2011.
5. A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. Accurate Invariant Checking for Programs Manipulating Lists and Arrays with Infinite Data. In *Proc. of ATVA'12*, *LNCS* 7561. Springer, 2012.
6. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular (Tree) Model Checking. *International Journal on Software Tools for Technology Transfer*, 14(2):167–191, 2012.
7. B.-Y. Chang, X. Rival, and G. Necula. Shape Analysis with Structural Invariant Checkers. In *Proc. of SAS'07*, *LNCS* 4634. Springer, 2007.
8. W.-N. Chin, C. David, H. Nguyen, and S. Qin. Automated Verification of Shape, Size and Bag Properties via User-defined Predicates in Separation Logic. *Science of Computer Programming*, 77(9):1006–1036, 2012.
9. K. Dudka, P. Peringer, and T. Vojnar. Byte-Precise Verification of Low-Level List Manipulation. In *Proc. of SAS'13*, *LNCS 7935*. Springer, 2013.
10. P. Habermehl, L. Holík, A. Rogalewicz, J. Šimáček, and T. Vojnar. Forest Automata for Verification of Heap Manipulation. In *Proc. of CAV'11*, *LNCS* 6806. Springer, 2011.
11. P. Habermehl, L. Holík, A. Rogalewicz, J. Šimáček, and T. Vojnar. Forest Automata for Verification of Heap Manipulation. *Formal Methods in System Design*, 41(1):83–106, 2012.
12. J. Heinen, T. Noll, and S. Rieger. Juggrnaut: Graph Grammar Abstraction for Unbounded Heap Structures. *ENTCS*, 266, 2010.
13. L. Holík, O. Lengál, A. Rogalewicz, J. Šimáček, and T. Vojnar. Fully Automated Shape Analysis Based on Forest Automata. In *Proc. of CAV'13*, LNCS 8044. Springer, 2013.
14. J. Jensen, M. Jørgensen, N. Klarlund, and M. Schwartzbach. Automatic Verification of Pointer Programs Using Monadic Second-order Logic. In *Proc. of PLDI'97*. ACM, 1997.
15. A. Loginov, T. Reps, and M. Sagiv. Abstraction Refinement via Inductive Learning. In *Proc. of CAV'05*, *LNCS* 3576. Springer, 2005.
16. S. Magill, M. Tsai, P. Lee, and Y.-K. Tsay. A Calculus of Atomic Actions. In *Proc. of POPL'10*, ACM, 2010.
17. W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *CACM*, 33(6), 1990.
18. S. Qin, G. He, C. Luo, W.-N. Chin, and X. Chen. Loop Invariant Synthesis in a Combined Abstract Domain. *Journal of Symbolic Computation*, 50, 2013.
19. S. Sagiv, T. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-valued Logic. *TOPLAS*, 24(3). ACM Press, 2002.
20. T. Wies, V. Kuncak, K. Zee, A. Podelski, and M. Rinard. On Verifying Complex Properties using Symbolic Shape Analysis. In *Proc. of HAV'07*, 2007.
21. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable Shape Analysis for Systems Code. In *Proc. of CAV'08*, *LNCS* 5123. Springer, 2008.
22. K. Zee, V. Kuncak, and M. Rinard. Full Functional Verification of Linked Data Structures. In *Proc. of PLDI'08*. ACM Press, 2008.

A Constraint Saturation

This appendix describes in a more detail the saturation procedure of Section 5.1. FA can be put into a saturated form by applying saturation rules, each of which infers new constraints from existing ones, until no more rules can be applied. Before describing our saturation rules, we first introduce some notation. For relations $\sim$ and $\sim'$ on $\mathbb{D}$, let $\sim \circ \sim'$ be the relation defined such that $d(\sim \circ \sim')d'$ iff there is a d'' such that $d \sim d''$ and $d'' \sim' d'$. We write $\sim \subseteq \sim'$ iff $d \sim d'$ implies $d \sim' d'$, and we define $\sim^{-1}$ by $d \sim^{-1} d'$ iff $d' \sim d$. We say that a constraint $q \sim'_{ry} q'$ is a *weakening* of a constraint $q \sim_{rx} q'$ iff $\sim \subseteq \sim'$ and $y = a$ implies $x = a$. The saturation rules that can be used are as follows.

- New global constraints can be inferred from existing global constraints by using properties of relations. In particular, one can use:
  - Transitivity to infer infer $q(\sim \circ \sim')_{ry}q''$ from $q \sim_{rx} q'$ and $q' \sim'_{ry} q''$ for $x, y \in \{r, a\}$ and any states q, q', q''.
  - Reflexivity to infer $q \preceq_{rr} q$ for any state q.
- New global or local constraints can be inferred by weakening the existing ones. In particular, from $q \sim_{ra} q'$, we infer the weaker constraint $q \sim_{rr} q'$, and from $q \prec_{rx} q'$, we infer $q \preceq_{rx} q'$ for any $x \in \{r, a\}$ and any states q, q' (and likewise for local constraints).
- Each local constraint $0 \prec_{rr} i$ where $q_i \in \Omega$ or q_i has nullary outgoing transitions only can be strengthened to $0 \prec_{ra} i$. The latter applies to global transitions too.
- Local constraints can be inferred from global ones (i.e., if $q \to \overline{a}(q_1, \ldots, q_m) : c$ is a transition, then a global constraint $q \sim_{rx} q_i$ adds $0 \sim_{rx} i$ to c; moreover, if q_i is a root reference (say, j), then a global constraint $q \sim_{rx} root(A_j)$ adds $0 \sim_{rx} i$ to c).
- If q is a state of a TA A and p is a state of A or another TA of the given FA such that in each sequence of states through which q can be reached from the root state of A there is a state q' such that $p \sim_{ra} q'$, then a constraint $p \sim_{ra} q$ is added as well.
- Whenever there is a TA A_1 with a root state q_0 and a state q such that (i) $q_0 \succeq_{rr} q$, (ii) q has an outgoing transition in whose right-hand side a state q_i appears where q_i is a reference to a TA A_2, and (iii) c includes a constraint $0 \succeq_{rr} i$, then a global constraint $q_0 \succeq_{rr} p_0$ can be added for the root state p_0 of A_2 (likewise for other kinds of relations than $\succeq_{rr}$). Conversely, from $q_0 \succeq_{rr} p_0$ and $q_0 \succeq_{rr} q$, one can derive the local constrain $0 \succeq_{rr} i$.
- Finally, global constraints can be inferred from existing ones by propagating them over local constraints of transitions in which the states of the global constraints occur. Since a single state may be reached in several different ways, propagation of global constraints through local constraints on all transitions arriving to the given state must be considered. If some of the ways how to get to the state does not allow the propagation, it cannot be done. Moreover, since one propagation can enable another one, the propagation must be done iteratively until a fixpoint is reached. The iterative propagation must terminate since the number of constraints that can be used is finite. The propagation of constraints between states of TA can be performed either downwards from the root towards leaves or upwards from leaves towards the root as described below.

Let p be the root state of some TA. For each state q of A, let $\Phi(q,p)$ be the set of global constraints between q and p. In the downwards propagation, we simultaneously extend the sets $\Phi(q,p)$ to larger ones $\Psi(q,p)$ provided that $\Psi(q,p) = \Phi(q,p)$ when q is the root state (i.e., no constraints are added at the root state) and provided that (when q is not the root state) for each constraint ϕ in $\Psi(q,p) \setminus \Phi(q,p)$ and each occurrence of q as q_i (say) in the right-hand side of a transition $\delta = q' \rightarrow \overline{a}(q_1,\ldots,q_m) : c$, either

- there is a local constraint $0 \sim'_{rr} i$ in c and a global constraint $q' \sim_{rx} p$ in $\Psi(q',p)$ with $x \in \{a,r\}$ such that ϕ is of the form $q\ ((\sim')^{-1} \circ \sim)_{rx}\ p$ or a weakening thereof,
- there is a local constraint $0 \sim'_{rx} i$ in c and a global constraint $p \sim_{ry} q'$ in $\Psi(q',p)$ with $x,y \in \{a,r\}$ such that ϕ is of the form $p\ (\sim \circ \sim')_{rx}\ q$ or a weakening thereof, or
- $p \sim_{ra} q'$ is in $\Psi(q',p)$ and ϕ is $p \sim_{ra} q$ or a weakening thereof.

Intuitively, the first two cases use transitivity to propagate a constraint involving q' to a constraint involving q_i; the last case uses the semantics of $p \sim_{ra} q'$.
The upwards propagation can be defined analogously. Already existing sets of constraints $\Phi(q,p)$ can be extended to sets $\Psi(q,p)$ provided that for each constraint ϕ in $\Psi(q,p) \setminus \Phi(q,p)$, either

- ϕ is of the form $p \sim_{ra} q$, the constraint $p \sim_{rr} q$ is in $\Psi(q,p)$, and $p \sim_{ra} q_i \in \Phi(q_i,p)$ for each i in $1,\ldots,m$ such that q_i is a state, or
- it is the case that a leaf of a tree can never be mapped to q in a run, and for each occurrence of q in the left-hand side of a transition $\delta = q \rightarrow \overline{a}(q_1,\ldots,q_m) : c$, either
  - there is a $0 \sim'_{rr} i$ in c and a constraint $q_i \sim_{rx} p$ in $\Psi(q_i,p)$ with $x \in \{a,r\}$ such that ϕ is of the form $q\ (\sim' \circ \sim)_{rx}\ p$ (or a weakening thereof), or
  - there is a $0 \sim'_{rr} i$ in c and a constraint $p \sim_{rx} q_i$ in $\Psi(q_i,p)$ with $x \in \{a,r\}$ such that ϕ is of the form $p\ (\sim \circ (\sim')^{-1})_{rr}\ q$ (or a weakening thereof).

B A Formal Description of Boxes

We introduce the so-called *nested forest automata* (NFA) which are FA that use as symbols both selectors and *boxes*. Boxes are NFA of a lower level that represent graphs with an *input port* and an *output port*. We will now extend the definitions from Section 4 to allow ports. For simplicity of presentation, we give only a simplified version of the definition in [10,11], which is more general and allows boxes with an arbitrary number of output ports.

Formally, we define an *io-graph* over $\langle \Gamma, \Omega \rangle$ to be a tuple $g_{io} = \langle g, i, o \rangle$ where g is a graph with two designated distinct nodes i and o called the *input* and *output port* respectively. An *io-forest* $(t_1 \cdots t_n)_{io}$ over $\langle \Gamma, \Omega \rangle$ is defined as $(t_1 \cdots t_n)_{io} = \langle t_1 \cdots t_n, i, o \rangle$ where $t_1 \cdots t_n$ is a forest and $i, o \in \{1,\ldots,n\}, i \neq o$, are the *input port* and *output port indices*. The composition operator $\otimes$ is extended to io-forests in the following way: $\otimes \langle t_1 \cdots t_n, i, o \rangle = \langle \otimes t_1 \cdots t_n, root(t_i), root(t_o) \rangle$, so the composition of an io-forest is an io-graph.

A *nested forest automaton* (NFA) over $\langle \Gamma, \Omega \rangle$ is an FA over $\langle \Gamma \cup \mathcal{B}, \Omega \rangle$ where $\mathcal{B}$ is a finite set of *boxes*. A *box* $\square$ over $\langle \Gamma, \Omega \rangle$, where Γ does not contain $\square$, is a triple

$\Box = \langle F_\Box, i, o\rangle$ such that $F_\Box$ is an NFA $F_\Box = \langle A_1 \cdots A_n, \varphi\rangle$ over $\langle \Gamma, \Omega\rangle$, $i \in \{1,\ldots,n\}$ is the *input port index*, and $o \in \{1,\ldots,n\}$ is the *output port index* such that $i \neq o$. The set of boxes of an NFA is required to form a hierarchy, i.e. a box cannot recursively contain itself. The *io-language* $L_{io}(\Box)$ of a box $\Box = \langle F_\Box, i, o\rangle$ is the set of io-graphs $L_{io}(\Box) = \{\otimes\langle t_1 \cdots t_n, i, o\rangle \mid t_1 \cdots t_n$ is accepted by $F_\Box\}$.

In the case of an NFA F, we need to distinguish between its language $L(F)$, which is a set of graphs over $\langle \Gamma \cup \mathcal{B}, \Omega\rangle$ and its *semantics*, which is a set of graphs over $\langle \Gamma, \Omega\rangle$ that emerges when all boxes in the graphs of the language are recursively *unfolded* in all possible ways. Formally, given a graph g, a graph g' is an *unfolding* of g (written as $g \leadsto g'$) if there is an occurrence $(u, \Box, v) \in next_g$ of a box $\Box$ in g (which may be seen as an edge from u to v over $\Box$ in g), such that g' can be constructed from g by substituting $(u, \Box, v)$ with $g_\Box$, which is done by removing $(u, \Box, v)$ from g, uniting g with $g_\Box$, and associating the input port of $g_\Box$ with u and the output port of $g_\Box$ with v, where $g_\Box \in L_{io}(\Box)$. We use $\leadsto^*$ to denote the reflexive transitive closure of $\leadsto$. The *semantics* of F, written as $[\![F]\!]$, is the set of all graphs g' over $\langle \Gamma, \Omega\rangle$ for which there is a graph g in $L(F)$ such that $g \leadsto^* g'$.

C Proofs for Section 5

Lemma 1. *The set of forests accepted by an FA F is equal to the union of decodings of forests accepted by F^E.*

Proof. The proof is technical but straightforward. Let $F = \langle A_1 \cdots A_n, \varphi\rangle$ and $F^E = \langle A_1^E \cdots A_n^E, \emptyset\rangle$. We first prove that every forest $t_1 \cdots t_n$ accepted by F belongs to the decoding of some forest accepted by F^E. Let $\rho_1, \ldots, \rho_n$ be the runs of $A_1 \cdots A_n$ on $t_1 \cdots t_n$, respectively. We will construct runs $\rho'_1, \ldots, \rho'_n$ of $A_1^E \ldots A_n^E$ on a forest $t'_1 \cdots t'_n$ of which $t_1 \cdots t_n$ is a decoding. For every $1 \leq i \leq n$, ρ'_i is constructed from ρ_i as follows. To simplify notation, let $\rho, t, \rho', t', A, A^E = \rho_i, t_i, \rho'_i, t'_i, A_i, A_i^E$. We let $V_{t'} = V_t$. $\lambda_{t'}$ can be chosen arbitrary. For every $v \in V_t$ s.t. $a_t^1(v) = v_1, \ldots a_t^m(v) = v_m$ are all edges of t with the source v, there is a transition of A of the form $\delta = q \to \overline{a}(q_1, \ldots, q_m) : c$ such that $\rho(v) = q$, $\rho(v_1) = q_1, \ldots, \rho(v_m) = q_m$, c is satisfied by $\lambda_t(v), \lambda_t(v_1), \ldots, \lambda_t(v_m)$, and moreover, global constraints $q \sim r, r \sim q \in \varphi$ where r is the root state of the k-th tree for some $1 \leq k \leq n$ are satisfied by the root of the k-th tree and v (by the definition of a run). ρ' then labels $v, v_1, \ldots, v_m$ using the rule $\delta' = q \to \bar{\alpha}(q_1, \ldots, q_m) : \emptyset$ which is obtained by the opration encoding of Section 5.4 from δ ($\bar{\alpha} = \langle (a_1, c_1, c_g) \cdots (a_m, c_m, c_g)\rangle$ where c_g contains encoded part of φ involving q and $c = c_1 \cup \ldots \cup c_m$). ρ' is obviously a run of A^E, so we are indeed constructing a sequence of runs $\rho'_1, \ldots, \rho'_n$ of $A_1^E \ldots A_n^E$. The construction of ρ' defines a map f which assigns to every $v, v_1, \ldots, v_m \in V_t$ where $v_1, \ldots, v_m$ are the successors of v a pair of transition (δ, δ') of A and A', respectively. δ and δ' are the rules used within ρ and ρ', respectively, to label $v, v_1, \ldots, v_n$.

To observe that $t_1 \cdots t_n$ is indeed a decoding of $t'_1 \cdots t'_n$, notice that for all $1 \leq i \leq n$, $V_{t_i} = V_{t'_i}$ and for every $v \in V_{t_i}$, and its successors $v_1, \ldots, v_m$ with $f(v, v_1, \ldots, v_m) = (\delta, \delta')$, the following holds: let $\delta = q \to \overline{a}(q_1, \ldots, q_m) : c$ and $\delta' = q \to \bar{\alpha}(q_1, \ldots, q_m) : \emptyset$ where $\bar{\alpha} = \langle (a_1, c_1, c_g) \cdots (a_m, c_m, c_g)\rangle$. Then (1) δ' specifies that $\alpha_1(v) = v, \ldots, \alpha_m(v) = v_m$ which means that the decoding function of Sec. 5.4 will define $a_1(v) = v, \ldots, a_m(v) =$

v_m. This is indeed the same as specified by δ for t. (2) the constraints imposed on the data value of v by δ and by φ within $\rho_1,\ldots,\rho_n$ are the same as the constraints imposed on v by the decoding function of Section 5.4 when applied on v within $t'_1\cdots t'_n$. In detail, c contains a local constraint $0\sim k$ iff c_k contains $0\sim k$ (by the def. of encoding). This means that in the run of A on t, it is required that $v\sim v_k$, which is the same constraint as required by the decoding function. Symmetrically for local constraints $k\sim 0$. Second, there is a global constrain $q\sim r\in\varphi$ s.t. r it the root state of A_k not appearing within right-hand sides iff $0\sim k\in c_g$ (by the def. of encoding). Then, in the run of A, this enforces that $v\sim u$ where u is the root of t_k. Notice that u cannot be any other node then the root since r does not appear within right-hand sides of rules and can thus accept only the root of t_k. These is precisely the same requirement as imposed by decoding based on the constraint $0\sim k$.

Second, we prove that the decoding of every forest $t'_1\cdots t'_n$ accepted by F^E contains only forests accepted by F. Let $\rho'_1,\ldots,\rho'_n$ be the runs of $A^E_1\ldots A^E_n$ on $t'_1\cdots t'_n$, respectively. For every forest $t_1\cdots t_n$ in the decoding of $t'_1\cdots t'_n$, we will construct runs $\rho_1,\ldots,\rho_n$ of $A_1\ldots A_n$. For every $1\le i\le n$, ρ_i simply equals ρ'_i (this definition is possible since t_i and t'_i have the same sets of nodes). We now need to check that $\rho_1,\ldots,\rho_n$ is indeed a sequence of runs of $A_1\cdots A_n$ accepting $t_1\cdots t_n$.

Let $1\le i\le n$, and to simplify notation, let $\rho,t,\rho',t',A,A^E=\rho_i,t_i,\rho'_i,t'_i,A_i,A^E_i$. Let $v\in V'_t$ s.t. $\alpha^1_{t'}(v)=v_1,\ldots\alpha^m_{t'}(v)=v_m$ are all edges of t' with the source v, and $\alpha_j=(a_j,c_j,c_g)$ for all $1\le j\le m$. By the definition of decoding, v satisfies all constraints encoded within $\bar\alpha$. Since $t'_1\cdots t'_n$ is accepted by F^E, there is a transition of A^E of the form $\delta'=q\to\bar\alpha(q_1,\ldots,q_m):\emptyset$ such that $\rho'(v)=q$, $\rho(v_1)=q_1,\ldots,\rho'(v_m)=q_m$. By the definition of encoding, δ' was created from a rule $\delta=q\to\bar a(q_1,\ldots,q_m):c$ of A where $c_1,\ldots,c_m$ encode c and c_g encode all global constraints involving q and a root state r which does not appear within a right-hand side of a rule. These are precisely the constraints mentioned above which are required to hold for v in $t_1\cdots t_n$ by decoding, hence they are in $t_1\cdots t_n$ satisfied. ρ is thus indeed a run of A compatible with $t_1\cdots t_n$ since for every v and its children $v_1,\ldots,v_m$, there is a suitable rule δ. □

Lemma 2. *Given two FAs F_A and F_B, $L(F^E_A)\subseteq L(F^{E^+}_B)\implies L(F_A)\subseteq L(F_B)$*

Proof (sketch). The statement is obviously true. Since the transformation from F^E_2 to $F^{E^+}_2$ adds only versions of existing rules encoding stronger constraints, the sets of decodings of forest of $F^{E^+}_2$ is the same the set of decodings of forests of F^E_2. The statement then follows immediately from Lemma 1. □

D Source Code of the Considered Case Studies

This appendix provides all source code examples that are actually analyzed by our framework.

D.1 Singly-Linked Lists

```
/*********************************************************************
```

```
   Name: Node
   Description: Structure of a node in a singly-linked list
**************************************************************************/
    struct Node {
        struct Node* next; // Next pointer field
        int data;          // Data field
    };
/*************************************************************************
   Name: insert
   Description: Insert a node into a singly-linked sorted list.
   Input: A head pointer 'head' of the list and the new node pointer 'new'.
   Output: New head pointer 'head' of the new list after inserting
**************************************************************************/
struct Node* insert(struct Node* new, struct Node* head){
             struct Node* curr = head;
             struct Node* prev = NULL;
             // Search for position of the new node
             while(curr != NULL && new->data > curr->data){
                   prev = curr;
                   curr = curr->next;
             }
             // If the new node is inserted after head node
             if(prev != NULL){
                new->next = curr;
                prev->next = new;
             }
             // If the new node is inserted before head node
             else{
                new->next = curr;
                // new node will be the new head node
                return new;
             }
             return head;
}
/*************************************************************************
   Name: remove
   Description:  Remove a node from a sorted list
   Input: A head pointer 'head' of the list and the value v of node that
          is removed
   Output: New head pointer 'head' of the new list after removing
**************************************************************************/
struct Node* remove(int v, struct Node* head){
            struct Node* curr = head;
            struct Node* pred = NULL;
            while (curr != NULL){
                // if current node is the removed node
                if (curr->data == v){
                    // if current node is not head node
                    if (pred)
                        pred->next = curr->next;
                    else
                        // if current node is head node
                        head = head->next;
                    free(curr);
                    break;
                }
                // if the removed node is not not found yet
                if(curr->data < v){
                   pred = curr;
                   curr = curr->next;
                }
                else
                   // there is no node in the list with value v
                   return head;
            }
            return head;
}
/*************************************************************************
   Name: reverse
   Description:  Reverse a sorted list
   Input: A head pointer 'head' of the list
   Output: New head pointer 'head' of the new list after reversing
*************************************************************************/
struct Node* reverse(struct T* head){
       struct Node* z = NULL;
       struct Node* y = NULL;
       while(head != NULL){
           y = head;
           head = head->next;
           y->next = z;
           z = y;
       }
       return y;
}
/*************************************************************************
   Name: insertionsort
   Description:  Sort a singly-linked list by insertion sort algorithm
   Input: A head pointer 'head' of the list
   Output: New head pointer 'head' of the sorted list
*************************************************************************/
struct Node* insertionsort(struct Node* head){
        // sorted is pointer to a sorted list
        struct Node* sorted = NULL;
        struct Node* pred = NULL;
        struct Node* z = NULL;
        struct Node* x = head;
        struct Node* y = head;
        while(x){
                y = x;
                x = x->next;
                pred = NULL;
                // find the position of y in the sorted list
                z = sorted;
                while (z && y->data > z->data){
                        pred = z;
                        z = z->next;
                }
                // insert y into the sorted list
                y->next = z;
                if(pred) pred->next = y;
                else
                  // update sorted pointer
                  sorted = y;
        }
        return sorted;
}
/*************************************************************************
   Name: bubblesort
   Description:  Sort a singly-linked list by bubblesort algorithm
   Input: A head pointer 'head' of the list
   Output: New head pointer 'head' of the sorted list
*************************************************************************/
struct Node* bubblesort(struct Node* head){
        struct Node* pred = NULL;
        struct Node* succ = NULL;
        struct Node* t = NULL;
        bool sorted = false;
        // while a list is not sorted
        while(!sorted){
              sorted = true;
              succ = head;
              while(succ->next != NULL){
                    pred = succ;
                    succ = succ->next;
                    // if a swap between two neighbour nodes is needed
                    if(pred->data > succ->data){
                       // swap
                       pred->next = succ->next;
                       succ->next = pred;
                       if(t)
                          t->next = succ;
                       else
                          head = succ;
                       t = pred;
                       pred = succ;
                       succ = t;
                       // update sorted to false
                       sorted = false;
                    }
                    t = pred;
                }
                t = NULL;
        }
        return head;
}
```

D.2 Doubly-Linked Lists

```
/************************************************************************
   Name: Node
   Description: Structure of a node in a doubly-linked list
************************************************************************/
    struct Node {
        struct Node* next; // Next pointer field
        struct Node* prev; // Prev pointer field
        int data;          // Data field
    };
/************************************************************************
   Name: insert
   Description: Insert a node into a sorted list.
   Input: A head pointer 'head' of the list and the new node pointer 'new'.
   Output: New head pointer 'head' of the new list after inserting
************************************************************************/
struct Node* insert(struct Node* new, struct Node* head){
        struct Node* curr = head;
        struct Node* prev = NULL;
        // search for position of inserted node
        while(curr != NULL && new->data > curr->data){
              prev = curr;
              curr = curr->next;
        }
        // if the inserted node is insert at the middle of the list
        if(prev != NULL && curr != NULL){
             prev->next = new;
             curr->prev = new;
             new->prev = prev;
             new->next = curr;
        }
        else {
               // if the inserted node is inserted at tail of the list
               if(prev != NULL){
                  prev->next = new;
                  new->prev = prev;
                  new->next = curr;
               }
               else{
                // if the inserted node is inserted at the head of the list
                 new->next = curr;
                 curr->prev = new;
                 new->prev = NULL;
                head = new;
             }
       }
       return head;
}
/*********************************************************************
   Name: remove
   Description:  Remove a node from a sorted list
   Input: A head pointer 'head' of the list and the value v of node that
          is removed
   Output: New head pointer 'head' of the new list after removing
*********************************************************************/
struct Node* remove(int v, struct Node* head){
    struct Node* curr = head;
    struct Node* pred = NULL;
    while(curr != NULL){
          // if current node is node we need to remove
          if (curr->data == v){
             // current node is not head node
             if (pred){
                 pred->next = curr->next;
                 if(curr->next)
                    curr->next->prev = pred;
             }
             else
                 // if current node is head node then head is updated
                 head = head->next;
                 free(curr);
                 break;
             }
             // if removed node is not found yet
             if(curr->data < v){
                  pred = curr;
                  curr = curr->next;
             }
             else
                // if there is node node with value v
                return head;
    }
    return head;
}
/*********************************************************************
   Name: reverse
   Description:  Reverse a sorted list
   Input: A head pointer 'head' of the list
   Output: New head pointer 'head' of the new list after reversing
*********************************************************************/
struct Node* reverse(struct T* head){
       struct Node* x = NULL;
       while(head != NULL){
           struct Node* z = head;
           head = head->next;
           if(head)
             head->prev = NULL;
           z->next = x;
           z->prev = NULL;
           if(x)
             x->prev = z;
           x = z;
       }
       return x;
}
/*************************************************************************
   Name: insertionsort
   Description:  Sort a linked list by insertion sort algorithm
   Input: A head pointer 'head' of the list
   Output: New head pointer 'head' of the sorted list
*************************************************************************/
struct Node* insertionsort(struct Node* head){
        // 'sorted' is pointer to a sorted list
        struct Node* sorted = NULL;
        struct Node* pred = NULL;
        struct Node* z = NULL;
        struct Node* x = head;
        struct Node* y = NULL;
        while(x){
                y = x;
                x = x->next;
                if(x) x->prev = NULL;
                // find the position of y in the sorted list
                z = sorted;
                pred = NULL;
                while (z && y->data > z->data){
                        pred = z;
                        z = z->next;
                }
                // insert y into the sorted list
                y->next = z;
                if(z) z->prev = y;
                y->prev = pred;
                if(pred) pred->next = y;
                else sorted = y;
        }
        return sorted;
}
/*************************************************************************
   Name: bubblesort
   Description:  Sort a singly-linked list by bubble sort algorithm
   Input: A head pointer 'head' of the list
   Output: New head pointer 'head' of the sorted list
```

```
**********************************************************************/
struct Node* bubblesort(struct Node* head){
        struct Node* pred = NULL;
        struct Node* succ = NULL;
        struct Node* t = NULL;
        bool sorted = false;
        // when the list has not been sorted yet
        while(!sorted){
                sorted = true;
                succ = head;
                while(succ->next){
                      pred = succ;
                      succ = succ->next;
                      // if its needed to swap two neighbour nodes
                      if(pred->data > succ->data){
                         // swap
                         if(succ->next){
                            pred->next = succ->next;
                            succ->next->prev = pred;
                         }
                         else
                            pred->next = NULL;
                         succ->next = pred;
                         pred->prev = succ;
                         if(t){
                            t->next = succ;
                            succ->prev = t;
                         }
                         else{
                             head = succ;
                             head->prev = NULL;
                         }
                         t = pred;
                         pred = succ;
                         succ = t;
                         /* update sorted variable to false
                         because the list is not sorted*/
                         sorted = false;
                      }
                      t = pred;
                }
                t = NULL;
        }
        return head;
}
```

D.3 Skip Lists with Two Levels

```
/**********************************************************************
   Name: Node
   Description: Structure of a node in a 2-pointer skip list
*********************************************************************/
    struct Node {
        struct Node* n1; // pointer field at higher level
        struct Node* n2  // pointer field at lower level
        int data;        // Data field
    };
/********************************************************************
   Name: insert
   Description: Insert a node into a skip list.
   Input: A head pointer 'head' and tail pointer 'tail'.
   Output: Return true if the node is inserted into the list, otherwise
           the function return false
*********************************************************************/
bool insert(struct Node* newNode, struct Node* head, struct Node* tail) {
     struct Node* sPred, sSucc,mPred, mSucc, pred, curr;
     pred = head ;
     curr = pred->n1;
     // in the case that new node is in data range of the list
     if(newNode->data < tail->data && head->data < newNode->data){
       // search for position at higher level
        while(newNode->data > curr->data){
            pred = curr ;
            curr = pred->n1 ;
        }
        sPred = pred;
        sSucc = curr;
        curr = pred->n2;
        // search for position at lower level
        while(newNode->data > curr->data){
             pred = curr ;
             curr = pred->n2 ;
        }
        mPred = pred;
        mSucc = curr;
        /* if the value of new node is not equal to value of any node
        in the list*/
        if(newNode->data != mSucc->data && newNode->data != sSucc->data){
           // insert at lower level
           newNode->n2 = mSucc;
           mPred->n2 = newNode ;
           // insert at higher level non-deterministically
           if(__nondet()){
              newNode->n1 = sSucc;
              sPred->n1 = newNode ;
           }
           // finish the inserting and return true
           return true;
        }
        else
```

```
           // if the value of new node is found in the list
           return false;
     }
     else
        // if the value of new node is out of the data range of the list
        return false;
}
/*************************************************************************
   Name: remove
   Description:  Remove a node from a skip list
   Input: A head pointer 'head' and tail pointer 'tail'.
   Output: Return true if the node with value v is removed from the list,
           otherwise the function return false
*************************************************************************/
bool remove(int v, struct Node* head, struct Node* tail) {
       struct Node* sPred, sSucc, mPred, mSucc, pred, curr;
       pred = head;
       curr = pred->n1;
       // in the case that v is in data range of the list
       if(tail->data > v && head->data < v){
         // search for position at higher level
         while (curr->data < v){
              pred = curr;
              curr = pred->n1;
         }
         sPred = pred;
         sSucc = curr;
         curr = pred->n2;
         // search for position at lower level
         while(curr->data < v){
           pred = curr ;
           curr = pred->n2 ;
         }
         mPred = pred;
         mSucc = curr;
         // if the node with data value v is found at the higher level
         if(sSucc->data == v){
             sPred->n1 = sSucc->n1;
             mPred->n2 = mSucc->n2;
             free(sSucc);
             return true;
         }
         else{
             // if the node with data value v is found at the lower level
             if(mSucc->data == v){
                mPred->n2 = mSucc->n2;
                free(mSucc);
                return true;
             }
             else
```

```
                // if the node with data value v is not found in the list
                return false;
        }
      }
      else
        return false;
}
```

D.4 Skip Lists with Three Levels

```
/**********************************************************************
   Name: Node
   Description: Structure of a node in a 3-pointer skip list
***********************************************************************/
    struct Node {
        struct Node* n1; // pointer field at top level
        struct Node* n2  // pointer field at sub level
        struct Node* n3  // pointer field at main level
        int data;        // Data field
    };
/**********************************************************************
   Name: insert
   Description: Insert a node into a skip list.
   Input: A head pointer 'head' and tail pointer 'tail'.
   Output: Return true if the node is inserted into the list, otherwise
           the function return false
***********************************************************************/
bool insert(struct Node* newNode, struct Node* head, struct Node* tail) {
     struct Node* tPred, tSucc, sPred, sSucc, mPred, mSucc, pred, curr;
     pred = head ;
     curr = pred->n1;
     // in the case that new node is in data range of the list
     if(newNode->data < tail->data && head->data < newNode->data){
       // search for position at top level
        while(newNode->data > curr->data){
            pred = curr ;
            curr = pred->n1 ;
        }
        tPred = pred;
        tSucc = curr;
        curr = pred->n2;
        // search for position at sub level
        while(newNode->data > curr->data){
             pred = curr ;
             curr = pred->n2 ;
        }
        sPred = pred;
        sSucc = curr;
        curr = pred->n3;
        // search for position at main level
        while(newNode->data > curr->data){
             pred = curr ;
             curr = pred->n3;
        }
        mPred = pred;
        mSucc = curr;
        /* if the value of new node is not equal to value of any node
        in the list*/
        if(newNode->data != tSucc->data && newNode->data != mSucc->data
           && newNode->data != sSucc->data){
            // randomly insert the new node at any level of skiplist
            newNode->n3 = mSucc; mPred->n3 = newNode;
            if(__nondet()){
               newNode->n2 = sSucc; sPred->n2 = newNode ;
               if(__nondet())
                  newNode->n1 = tSucc; tPred->n1 = newNode ;
            }
           // finish the inserting and return true
           return true;
        }
        else
           // if the value of new node is found in the list
           return false;
     }
     else
        // if the value of new node is out of the data range of the list
        return false;
}
/*************************************************************************
   Name: remove
   Description:  Remove a node from a skip list
   Input: A head pointer 'head' and tail pointer 'tail'.
   Output: Return true if the node with value v is removed from the list,
           otherwise the function return false
*************************************************************************/
bool remove(int v, struct Node* head, struct Node* tail) {
      struct Node* tPred, tSucc, sPred, sSucc, mPred, mSucc, pred, curr;
      pred = head;
      curr = pred->n1;
      // in the case that v is in data range of the list
      if(tail->data > v && head->data < v){
         // search for position at top level
         while (curr->data < v){
              pred = curr;
              curr = pred->n1;
         }
         tPred = pred;
         tSucc = curr;
         curr = pred->n2;
         // search for position at sub level
         while(curr->data < v){
           pred = curr ;
           curr = pred->n2 ;
         }
         sPred = pred;
         sSucc = curr;
         curr = pred->n3;
         // search for position at main level
         while(curr->data < v){
           pred = curr ;
           curr = pred->n3 ;
         }
         mPred = pred;
         mSucc = curr;
         // if the node with data value v is found at the top level
         if(tSucc->data == v){
             tPred->n1 = tSucc->n1;
             sPred->n2 = sSucc->n2;
             mPred->n3 = mSucc->n3;
             free(tSucc);
             return true;
         }
         else{
            // if the node with data value v is found at the sub level
            if(sSucc->data == v){
               sPred->n2 = sSucc->n2;
               mPred->n3 = mSucc->n3;
               free(sSucc);
               return true;
            }
            else{
               // if the node with data value v is found at the main level
               if(mSucc->data == v){
                  mPred->n3 = mSucc->n3;
                  free(mSucc);
                  return true;
               }
               else
                 // if the node with data value v is not found in the list
                 return false;
            }
         }
      }
      else
        return false;
}
```

D.5 Binary Search Trees

```
/**********************************************************************
   Name: Node
   Description: Structure of a node in a binary search tree
**********************************************************************/
    struct Node {
        struct Node* left;  // Left pointer field
        struct Node* right; // Right pointer field
        int data;           // Data field
    };
/**********************************************************************
   Name: insert
   Description: Insert a node into a binary search tree.
   Input: A root pointer 'root' of the tree and the new node pointer 'new'.
   Output: New root pointer 'root' of the new tree after inserting
**********************************************************************/
struct Node* insert(struct Node* root, struct Node* new){
                 Node *x;
                 if(!root) return new;
                 x = root;
                 while(x.data != new.data)
                   if(x.data < new.data)
                       if (x.right) x = x.right;
                       else x.right = new;
                   else
                       if (x.left) x = x.left;
                       else x.left = new;
                 return root;
}
/**********************************************************************
   Name: leftRotate
   Description: Left rotate a binary search tree.
   Input: A root pointer 'root' of the tree.
   Output: New root pointer 'root' of the new tree after left rotating
**********************************************************************/
struct Node* leftRotate(struct Node* root){
 if(root->left){
     struct Node* r = root->left;
     root->left = r->right;
     r->right = root;
     return r;
     }
  return root;
}
/**********************************************************************
   Name: rightRotate
   Description: Right rotate a binary search tree.
   Input: A root pointer 'root' of the tree.
   Output: New root pointer 'root' of the new tree after right rotating
```

```
*************************************************************************/
struct Node* rightRotate(struct Node* root){
 if(root->right){
       struct Node* r = root->right;
       root->right = r->left;
       r->left = root;
     return r;
     }
  return root;
}
/************************************************************************
   Name: remove
   Description: Remove a node from a binary search tree.
   Input: A root pointer 'root' of the tree and value v of the node we need
          to remove
   Output: New binary search tree
*************************************************************************/
void remove(int v, struct Node* root){
      struct Node* t;
      struct Node* right = NULL;
      struct Node* sparent = NULL;
      struct Node* min = NULL;
      struct Node* node = root;
      struct Node* parent = root;
      // search for the node with value v
      while(node->data != v) {
            parent = node;
            if (parent->data < v){
                if(node->left)
                   node = node->left;
                else
                  break;
            }
            else {
                if(node->right)
                   node = node->right;
                else
                  break;
            }
     }
     t = node;
     // only delete the node with value v and its not a root node
     if(node != root && node->data == v){
      // if the removed node does not have right subtree
      if(t->right == NULL){
           node = node->left;
           free(t);
     }
     else{
          /* if the right subtree of the removed node
```

```
             does not have left subtree*/
          if (t->right->left == NULL){
             node = node->right; node->left = t->left; free(t);
          }
          /* if the right subtree of the removed node
            has left subtree*/
          else{
               right = t->right;
               min = t->right->left;
               sparent = right;
               /* search for the smallest node pointed by 'min'
                  in the right subtree*/
               while(min->left != NULL){
                  sparent = min;
                  min = min->left;
               }
               // exchange removed node with the smallest node
               node = min;
               sparent->left = min->right;
               min->left = t->left;
               min->right = right;
               free(t);
         }
     }
     if(parent->data < v) parent->left = node;
     else parent->right = node;
    }
}
```