Noname manuscript No.
(will be inserted by the editor)

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

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

Received: date / Accepted: date

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 with 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.

Keywords forest automata · shape analysis · dynamic linked data structures · tree automata · abstraction

This paper is an extended version of a paper published in the proceedings of ATVA'13. The work was supported by the Czech Science Foundation (projects 14-11384S and 13-37876P), the internal BUT project FIT-S-14-2486, 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.

P.A. Abdulla, B. Jonsson, and C.Q. Trinh
Uppsala University, Department of Information Technology
Box 337, 751 05 Uppsala, Sweden
E-mail: {parosh,bengt,cong-quy.trinh}@it.uu.se

L. Holík, O. Lengál, and T. Vojnar
FIT, Brno University of Technology, IT4Innovations Centre of Excellence,
Božetěchova 2, 61266 Brno, Czech Republic
E-mail: {holik,ilengal,vojnar}@fit.vutbr.cz

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 section).

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 [15]. 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 FAs, employing the framework of *abstract regular tree model checking* (ARTMC) [8], has been implemented in the Forester tool [17]. 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 [23], 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 TAs 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 TAs and capture relationships between data in distant parts of the heap. In order to obtain a powerful analysis based on such extended FAs, the entire analysis machinery must be redesigned, including a need to develop mechanisms for propagating data constraints through FAs, 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, like sortedness, can be checked by adding code that checks pre- and post-conditions. Functional properties can be checked by querying the computed over-approximation of the set of reachable configurations as well.

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, like 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 [17], but only after modifying the code in such a way that the preservation of the shape invariant does not depend on ordering relations.

This paper is an extension of the work originally published in [3]. In addition to what was presented in that work, we provide missing proofs and cover the related work in more detail. Furthermore, to make the presentation easier to comprehend, we provide more examples and more detailed descriptions of the concepts in the sections on constraint saturation, abstract transformers, and boxes.

Outline. After a review of related work, in Section 3, we present our approach to 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 directions for future work.

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 [15, 17]. 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 FAs have been used, including, e.g., separation logic and various related graph formalisms [29, 21, 10, 14], other logics [26, 18], automata [8], or graph grammars [16]. Compared with FAs, these approaches typically handle less general heap structures (often restricted to various classes of lists) [29, 14], they are less automated (requiring the user to specify loop invariants [18] or at least inductive definitions of the involved data structures [21, 10, 16]), or less scalable [8].

Verification of properties depending on the ordering of data stored in SLLs was considered in [6], 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 [6] 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 FAs. In order to achieve this, we had to not only propose a suitable way of combining ordering relations with FAs, but we also had to significantly modify many of the operations used over FAs.

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 [5], 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 [20] as well. Unlike our approach, [20] 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 [20] 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 [20]). A diffent aproach was taken in [4], where the TVLA framework is combined with predicate abstraction implemented in BLAST. The approach was experimentally run on several list-manipulating programs only.

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 [12]. 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 [24], but a need to manually provide inductive shape predicates remains.

The work considered in [9] extends the previous work [10] with data constraints. The method still needs shape invariants extended with data to be provided manually. The join and widening operations used on the shape level are extended with subsequent join and widening on the data level to cope with the data during the analysis.

Another work that targets verification of programs with dynamic linked data structures, including properties depending on the data stored in them, is [30]. 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 [27]. 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

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

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

some of the needed abstraction predicates. A further extension of this approach given in [28] increases the degree of automation and synthesizes the loop invariants automatically using counterexample guided refinement.

Several works, including [7], 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. Other approaches presenting various logical fragments for reasoning about heaps and the data stored in them together with decision procedures of these fragments were presented, e.g., in [22,25,11,19]. None of these approaches has been extended to a fully automatic verification method.

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. 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.

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 $\perp$ representing the `NULL` value, 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.

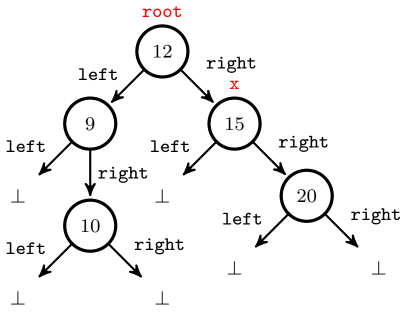
Fig. 2(a) shows a possible heap of the program in Fig. 1. Nodes are shown as circles, labeled by their data values. Selectors are shown as edges. Each selector points either to a node or to $\perp$ (denoting `NULL`). Some nodes are labeled by a pointer variable that points to them. The node with data 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 $\overline{2}$ ($\bar{\imath}$ denotes a reference to the i-th tree t_i) to indicate that in the graph, it points to the corresponding cut-point.

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 $\perp$; in tree components, Ω will also include root references.

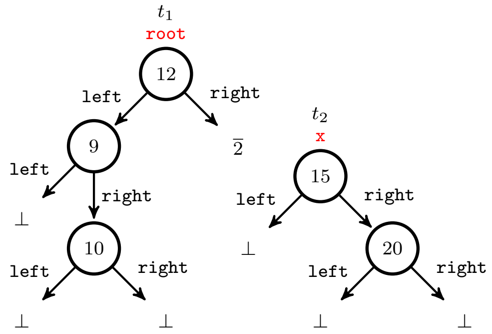
We use $f : A \rightharpoonup B$ to denote a partial function from A to B (also viewed as a total function $f : A \to (B \cup \{\top\})$, assuming that $\top \notin B$). We assume an unbounded data domain $\mathbb{D}$ with a total ordering relation $\preceq$.

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 $\langle \Gamma, \{\perp\} \rangle$ where $\perp$ 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 $V_g \cup \{\perp\}$ 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



(a) A graph.



(b) A forest decomposition.

Fig. 2 Decomposition of a graph into trees.

is given by a standard mapping from configurations to their successors, which we omit here.

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 elements 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 $\langle \Gamma, \{\perp\}\rangle$, 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 $v \in V_{t_k}$. We will use the following notation to talk about relations of data values of nodes within a forest. Given nodes u, v of trees t, t', respectively, of a forest and a relation $\sim \in \{\prec, \preceq, =, \succ, \succeq\}$, we denote by $u \sim_{\mathtt{rr}} v$ that $\lambda_t(u) \sim \lambda_{t'}(v)$ and we denote by $u \sim_{\mathtt{ra}} v$ that $\lambda_t(u) \sim \lambda_{t'}(w)$ for all nodes w in the subtree of t' rooted at v. We call these two types of relationships *root-root* and *root-all* relations, respectively.

4 Forest Automata

A forest automaton is essentially a tuple of tree automata accepting a set of tuples of composable 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 $0 \sim_{\mathtt{r}x} i$ where $\sim \in \{\prec, \preceq, \succ, \succeq\}$ (with $=$ viewed as syntactic sugar[1]), $i \in \{1, \dots, m\}$, and $x \in \{\mathtt{r}, \mathtt{a}\}$.

Intuitively, a local constraint of the form $0 \sim_{\mathtt{rr}} i$ associated with a transition of the form $q \to \overline{a}(q_1, \dots, q_m)$ of a TA $A = (Q, q_0, \Delta)$ states that, for each tree t' accepted by A at q_0, the data value of the *root* of the subtree t of t' that is accepted at q is related by $\sim$ with the data value of the *root* of the i-th subtree of t accepted at q_i. A local constraint of the form $0 \sim_{\mathtt{ra}} i$ states that, for each tree t' accepted by A, the data value of the *root* of the subtree t of t' that is 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 $q \to \overline{a}(q_1, \dots, q_m) : c$ in Δ with $\overline{a} = a^1 \cdots a^m$ such that (1) $\rho(v) = q$, (2) for all $1 \leq i \leq m$, 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 $0 \sim_{\mathtt{r}x} i$ in c where $x \in \{\mathtt{r}, \mathtt{a}\}$, it holds that $v \sim_{\mathtt{r}x} a^i_t(v)$. We define the *language* of A as $L(A) = \{t \mid \text{there is a run of } A \text{ over } t\}$.

Example 1 BSTs, such as the tree labeled by `root` but without the variable `x` in Fig. 2(a), are accepted by the TA over $\langle \Gamma, \Omega \rangle$ with one state q_1, which is also the root state (denoted by $\underline{q_1}$), and the following four transitions:

$$\underline{q_1} \to \mathtt{left}, \mathtt{right}(q_1, q_1) : 0 \succ_{\mathtt{ra}} 1, 0 \prec_{\mathtt{ra}} 2 \qquad \underline{q_1} \to \mathtt{left}, \mathtt{right}(q_1, \perp) : 0 \succ_{\mathtt{ra}} 1$$
$$\underline{q_1} \to \mathtt{left}, \mathtt{right}(\perp, q_1) : 0 \prec_{\mathtt{ra}} 2 \qquad \underline{q_1} \to \mathtt{left}, \mathtt{right}(\perp, \perp)$$

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. □

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 $n \geq 0$, is a sequence of TAs over $\langle \Gamma, \Omega \uplus \{\overline{1}, \dots, \overline{n}\} \rangle$ whose sets of states $Q_1, \dots, Q_n$ are mutually disjoint.
- φ is a set of *global data constraints* between the states of $A_1 \cdots A_n$, each of the form $q \sim_{\mathtt{rr}} q'$ or $q \sim_{\mathtt{ra}} q'$ where $q, q' \in \cup_{i=1}^n Q_i$, at least one of q, q' is a root state and $\sim \in \{\prec, \preceq, \succ, \succeq\}$ (with $=$ viewed as syntactic sugar). Intuitively, $q \sim_{\mathtt{r}x} q'$ says that for any two nodes v, v' in a forest accepted by q and q', respectively, data values must satisfy $v \sim_{\mathtt{r}x} v'$.

A forest $t_1 \cdots t_n$ over $\langle \Gamma, \Omega \rangle$ is *accepted* by F iff there are runs $\rho_1, \dots, \rho_n$ such that ρ_i is a run of A_i over t_i for every $1 \leq i \leq n$, and for each global constraint of the form $q \sim_{\mathtt{r}x} q'$ where $x \in \{\mathtt{r}, \mathtt{a}\}$, q is a state of some A_i and q' is a state of some A_j, we have $v \sim_{\mathtt{r}x} v'$ whenever $\rho_i(v) = q$ and $\rho_j(v') = q'$. 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 $\langle \Gamma, \{\perp\} \rangle$ represents a set of heaps H over Γ[2].

[1] The use of $\neq$ is forbidden because it would lead to a disjunction of constraints, which we do not support in this work.

[2] Note that from the definitions of languages of TAs and FAs, the effect of the $\sim_{\mathtt{ra}}$ data constraint (both local and global) is local to the TAs it is related to.

$$F = \langle A_1\, A_2, \varphi\rangle$$
$$\sigma(\texttt{root}) = 1, \sigma(\texttt{x}) = 2$$
$$A_1 : \begin{cases} \underline{q_{\mathtt{r}}} \to \texttt{left}, \texttt{right}(q_1, \overline{2}) \;\; : 0 \succ_{\mathtt{ra}} 1, 0 \prec_{\mathtt{ra}} 2 \\ q_1 \to \texttt{left}, \texttt{right}(\perp, q_2) : 0 \prec_{\mathtt{ra}} 2 \\ q_2 \to \texttt{left}, \texttt{right}(\perp, \perp) \end{cases}$$
$$A_2 : \begin{cases} \underline{q_{\mathtt{x}}} \to \texttt{left}, \texttt{right}(\perp, q_3) : 0 \prec_{\mathtt{ra}} 2 \\ q_3 \to \texttt{left}, \texttt{right}(\perp, \perp) \end{cases}$$
$$\varphi = \begin{Bmatrix} q_{\mathtt{x}} \succ_{\mathtt{ra}} q_{\mathtt{r}}, q_3 \succ_{\mathtt{ra}} q_{\mathtt{r}}, \\ q_{\mathtt{r}} \succ_{\mathtt{ra}} q_{\mathtt{x}}, q_1 \prec_{\mathtt{ra}} q_{\mathtt{x}}, q_2 \prec_{\mathtt{ra}} q_{\mathtt{x}} \end{Bmatrix}$$

Fig. 3 An example of an abstract configuration that is a possible representation of the concrete configuration shown in Fig. 2(b).

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 relate 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 FAs by checking inclusion component-wise on the languages of the component TAs.

5 FA-based Shape Analysis with Data

Our verification procedure performs a standard abstract interpretation [13]. The concrete domain in our case assigns to each program location a finite set of pairs $\langle\sigma, H\rangle$ where the *valuation* σ maps every variable to $\perp$, 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 $\perp$, an index of a TA in F, or to an undefined value, and F is an FA representing a set of heaps.

Example 2 Fig. 3 illustrates an abstract configuration $\langle\sigma, F\rangle$ that is a possible representation of the concrete configuration $\langle\sigma, H\rangle$ shown in Fig. 2(b). □

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 FAs in order to determine whether a fixpoint has been reached. Prior to checking language inclusion, we normalize the FAs, 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* [8]. The widening is applied to individual TAs 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., $q \sim_{\mathtt{r}x} p$ occurs as a global constraint if and only if $q' \sim_{\mathtt{r}x} p$ 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 [17]. 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 FAs 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. FAs 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 this section, we show the saturation rules that are used to deduce new data constraints from already existing ones. The saturation rules are used in a fixed point computation to deduce both global and local constraints from global constraints, local constraints, or their combinations.

Table 1 Rules for inferring global constraints from global constraints.

$$\frac{q \sim_{\mathtt{rr}} q' \qquad q' \sim'_{\mathtt{r}x} q''}{q\,(\sim \circ \sim')_{\mathtt{r}x}\, q''}\ \textsc{G-Trans}$$

$$\frac{}{q \simeq_{\mathtt{rr}} q}\ \textsc{G-Refl1} \qquad\qquad \frac{q' \sim_{\mathtt{rr}} q}{q \sim^{-1}_{\mathtt{rr}} q'}\ \textsc{G-Refl2}$$

$$\frac{q \sim_{\mathtt{rr}} q' \qquad \mathtt{Leaf}(q')}{q \sim_{\mathtt{ra}} q'}\ \textsc{G-Stre}$$

$$\frac{q \sim_{\mathtt{ra}} q'}{q \sim_{\mathtt{rr}} q'}\ \textsc{G-Weak1} \qquad\qquad \frac{q \sim_{\mathtt{r}x} q'}{q \simeq_{\mathtt{r}x} q'}\ \textsc{G-Weak2}$$

$$\frac{\mathtt{root}(A) \sim_{\mathtt{ra}} \mathtt{root}(A') \qquad q' \in Q(A')}{\mathtt{root}(A) \sim_{\mathtt{ra}} q'}\ \textsc{G-RootAll}$$

- $x \in \{r, a\}$
- $\simeq\, \in \{\preceq, \succeq\}$,
- $\sim \circ \sim'$ denotes the composition of $\sim$ and $\sim'$,
- $\mathtt{Leaf}(q)$ means that q has only nullary outgoing transitions or $q \in \Omega$,
- $Q(A')$ is the set of states of the TA A',
- $\mathtt{root}(A)$ is the root state of the TA A (likewise for A').

Before the description of the saturation rules, we first introduce some notation. For relations $\sim$ and $\sim'$ on $\mathbb{D}$, let $\sim \circ \sim'$ be the weakest relation from $\{\prec_{\mathtt{r}x}, \preceq_{\mathtt{r}x}, \succ_{\mathtt{r}x}, \succeq_{\mathtt{r}x}\}$, for $x \in \{\mathtt{r}, \mathtt{a}\}$, such that for all $d_1, d_2, d_3 \in \mathbb{D}$, it holds that $d_1 \sim d_2 \wedge d_2 \sim' d_3 \implies d_1\, (\sim \circ \sim')\, d_3$. 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 $q \sim'_{\mathtt{r}y} q'$ is a *weakening* of a constraint $q \sim_{\mathtt{r}x} q'$ iff it holds that $\sim\, \subseteq\, \sim'$ and, in the case y is $\mathtt{a}$ (i.e., a "for all" constraint), it also holds that x is $\mathtt{a}$. The saturation rules that can be used are as follows.

5.1.1 Inferring global constraints from global constraints

The saturation rules for inferring global constraints from global constraints, as shown in Table 1, are based on the following principles:

1. properties of the ordering relations:
   - G-Trans is based on transitivity,
   - G-Refl1 and G-Refl2 are based on reflexivity of $\preceq$ and $\succeq$,
2. weakening of existing data constraints:
   - G-Weak1 states that from $q \sim_{\mathtt{ra}} q'$, we can infer a weaker constraint $q \sim_{\mathtt{rr}} q'$,
   - G-Weak2 gives a rule for inferring the weaker constraints $q \preceq_{\mathtt{r}x} q'$ from $q \prec_{\mathtt{r}x} q'$ and $q \succeq_{\mathtt{r}x} q'$ from $q \succ_{\mathtt{r}x} q'$ for any $x \in \{\mathtt{r}, \mathtt{a}\}$,
3. strengthening of existing data constraints:
   - G-Stre states that each global constraint $q \sim_{\mathtt{rr}} q'$ where $q' \in \Omega$ or q' has nullary outgoing transitions only can be strengthened to $q \prec_{\mathtt{ra}} q'$,
4. properties of the ra relation:

Table 2 Rules for inferring local constraints from local constraints.

$$\frac{0 \sim_{\mathtt{ra}} i \in c}{0 \sim_{\mathtt{rr}} i \in c}\ \textsc{L-RootRoot}$$

$$\frac{0 \sim_{\mathtt{r}x} i \in c}{0 \simeq_{\mathtt{r}x} i \in c}\ \textsc{L-Weak} \qquad \frac{0 \sim_{\mathtt{rr}} i \in c \qquad \mathtt{Leaf}(q_i)}{0 \sim_{\mathtt{ra}} i \in c}\ \textsc{L-Stre}$$

- We assume the transition $q \to \overline{a}(q_1, \ldots, q_m) : c$ and $1 \leq i \leq m$,
- $x \in \{r, a\}$,
- $\simeq\ \in \{\preceq, \succeq\}$,
- $\mathtt{Leaf}(q)$ is true iff q has only nullary outgoing transitions or $q \in \Omega$,
- $\mathtt{root}(A)$ is the root state of the TA A.

Table 3 Rules for inferring local constraints from global constraints.

$$\frac{q \sim_{\mathtt{r}x} q_i}{0 \sim_{\mathtt{r}x} i \in c}\ \textsc{L-G-Prop} \qquad \frac{q_i = j \in \Omega \qquad q \sim_{\mathtt{r}x} \mathtt{root}(A_j)}{0 \sim_{\mathtt{r}x} i \in c}\ \textsc{L-G-Ref}$$

- We assume the transition $q \to \overline{a}(q_1, \ldots, q_m) : c$ and $1 \leq i \leq m$,
- $x \in \{r, a\}$.

- G-RootAll states for a pair of TAs A and A' of the given FA that if q' is a state of a TA A', then a global constraint $\mathtt{root}(A) \sim_{\mathtt{ra}} \mathtt{root}(A')$ will add the constraint $\mathtt{root}(A) \sim_{\mathtt{ra}} q'$.

5.1.2 Inferring local constraints from local constraints

The saturation rules (shown in Table 2) which infer new local constraints from already existing ones in a transition $q \to \overline{a}(q_1, \ldots, q_m) : c$, s.t. $1 \leq i \leq m$, are based on the following:

1. weakening the existing constraints: if $q \to \overline{a}(q_1, \ldots, q_m) : c$ is a transition, then
   - L-RootRoot weakens a $\sim_{\mathtt{ra}}$ relation to a $\sim_{\mathtt{rr}}$ relation,
   - L-Weak infers the weaker constraints $0 \preceq_{\mathtt{r}x} i$ from $0 \prec_{\mathtt{r}x} i$ and $0 \succeq_{\mathtt{r}x} i$ from $0 \succ_{\mathtt{r}x} i$ for any $x \in \{\mathtt{r}, \mathtt{a}\}$,
2. strengthening of existing data constraints:
   - L-Stre is used for q_i such that q_i is either in Ω or has only nullary outgoing transitions to strengthen a constraint $0 \sim_{\mathtt{rr}} i$ to the constraint $0 \sim_{\mathtt{ra}} i$.

5.1.3 Inferring local constraints from global constraints

Inference of local constraints in a transition $q \to \overline{a}(q_1, \ldots, q_m) : c$, s.t. $1 \leq i \leq m$, from global constraints is done using the rules shown in Table 3:

- L-G-Prop propagates a global constraint $q \sim_{\mathtt{r}x} q_i$ for states used in the same transition into a local constraint $0 \sim_{\mathtt{r}x} i$,
- L-G-Ref propagates a global constraint $q \sim_{\mathtt{r}x} \mathtt{root}(j)$ between a state q and the root state of a TA j into a local constraint $0 \sim_{\mathtt{ra}} i$ between q and $q_i = j$.

5.1.4 Inferring global constraints from local constraints

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 a 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 A. For each state q of A, let $\Phi(q,p)$ be the set of global constraints between q and p. The data constraints are propagated in two directions:

Downward propagation. In the downward propagation, we simultaneously extend the sets $\Phi(q,p)$ to larger ones $\Psi(q,p)$ starting from the root state q_0 of A and setting $\Psi(q_0,p) = \Phi(q_0,p)$ (i.e. no constraints are added for this case). Then, for non-root states q, we extend the set of constraints in $\Psi(q,p)$ by traversing over the transitions of A and adding constraints according to the following rules:

- We add the constraint $q\ ((\sim')^{-1} \circ \sim)_{\mathtt{r}x}\ p$, with $x \in \{\mathtt{a},\mathtt{r}\}$, if, for every occurrence of q as q_i in any transition $\delta = q' \to \overline{a}(q_1,\dots,q_n) : c$, there is a local constraint $0 \sim'_{\mathtt{rr}} i$ in c and a global constraint $q' \sim_{\mathtt{r}x} p$ in $\Psi(q',p)$.
- We add the constraint $p\ (\sim \circ \sim')_{\mathtt{r}x}\ q$, with $x \in \{\mathtt{a},\mathtt{r}\}$, if, for every occurrence of q as q_i in any transition $\delta = q' \to \overline{a}(q_1,\dots,q_n) : c$, there is a local constraint $0 \sim'_{\mathtt{r}x} i$ in c and a global constraint $p \sim_{\mathtt{r}y} q'$ in $\Psi(q',p)$ with $y \in \{\mathtt{a},\mathtt{r}\}$.
- We add the constraint $p \sim_{\mathtt{ra}} q$ if, for every occurrence of q as q_i in any transition $\delta = q' \to \overline{a}(q_1,\dots,q_n) : c$, it holds that $p \sim_{\mathtt{ra}} q'$ is in $\Psi(q',p)$.

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 $p \sim_{\mathtt{ra}} q'$.

Upward propagation. The upward propagation can be defined analogously. Already existing sets of constraints $\Phi(q,p)$ can be extended to sets $\Psi(q,p)$ by traversing over the transitions of A and adding constraints according to the following rules:

- We add the constraint $p \sim_{\mathtt{ra}} q$ if there is the constraint $p \sim_{\mathtt{rr}} q$ is in $\Psi(q,p)$, and for every transition $\delta = q \to \overline{a}(q_1,\dots,q_n) : c$ it holds that $p \sim_{\mathtt{ra}} q_i \in \Psi(q_i,p)$ for every $1 \le i \le n$.
- We add the constraint $q\ (\sim' \circ \sim)_{\mathtt{r}x}\ p$, with $x \in \{\mathtt{a},\mathtt{r}\}$, if there is no nullary transition going from q and for every transition $\delta = q \to \overline{a}(q_1,\dots,q_n) : c$, there are the constraints $0 \sim'_{\mathtt{rr}} i$ in c and $q_i \sim_{\mathtt{r}x} p$ in $\Psi(q_i,p)$ for some $1 \le i \le n$.
- We add the constraint $p\ (\sim \circ (\sim')^{-1})_{\mathtt{rr}}\ q$, with $x \in \{\mathtt{a},\mathtt{r}\}$, if there is no nullary transition going from q and for every transition $\delta = q \to \overline{a}(q_1,\dots,q_n) : c$, there are the constraints $0 \sim'_{\mathtt{rr}} i$ in c and $p \sim_{\mathtt{r}x} q_i$ in $\Psi(q_i,p)$ for some $1 \le i \le n$.

Proposition 1 *The constraint saturation process always terminates.*

Proof Follows from the facts that the maximum number of constraints in an FA is finite and that adding a new constraint is a monotone operation. □

5.2 Abstract Transformers

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

Below, we present the abstract transformers corresponding to some of the operations on abstract states of the form $\langle \sigma, F \rangle$—the rest of the transformers is analogous. For simplicity of the presentation, we assume that for all TAs A_i in F, (a) the root state of A_i does not appear on 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. Indeed, in order to transform a TA $A = \langle Q, q_f, \Delta \rangle$ from an FA F into the form where q_f does not appear on the right-hand side of any transition and appears on the left-hand side of exactly one transition, we may perform the following sequence of actions:

1. create a copy q'_f of q_f, which replaces q_f on the right-hand side of all transitions,
2. duplicate all transitions from q_f to become transitions also from q'_f (while again substituting any occurrence of q_f with q'_f),
3. split A into several TAs, one for each transition from the accepting state q_f, creating several copies of the FA F that contains A, and
4. adapt the local and global constraints by duplicating them whenever some state is duplicated.

An example of this transformation, which basically unfolds once all loops on q_f, will be given in Example 3 below.

We now introduce some common notation and operations for the below presented transformers. We use $A_{\sigma(\mathtt{x})}$ and $A_{\sigma(\mathtt{y})}$ to denote the TA pointed by variables $\mathtt{x}$ and $\mathtt{y}$, respectively, and $q_{\mathtt{x}}$ and $q_{\mathtt{y}}$ to denote the root states of these TAs. Let $q_{\mathtt{y}} \to \overline{a}(q_1, \ldots, q_i, \ldots, q_m) : c$ be the unique transition from $q_{\mathtt{y}}$. Before describing the actual update, let us first define how to split a TA.

The operation of *splitting* a TA $A_{\sigma(\mathtt{y})}$ at the i-th position, for $1 \leq i \leq m$, is described by the following sequence of operations:

1. First, a new TA A_k is appended to F such that A_k is a copy of $A_{\sigma(\mathtt{y})}$ but with q_i as the root state.
2. Second, the root transition in $A_{\sigma(\mathtt{y})}$ is changed to $q_{\mathtt{y}} \to \overline{a}(q_1, \ldots, \overline{k}, \ldots, q_m) : c'$ where c' is obtained from c by replacing any local constraint of the form $0 \sim_{\mathtt{r}x} i$ by the global constraint $q_{\mathtt{y}} \sim_{\mathtt{r}x} root(A_k)$.
3. Global data constraints are adapted as follows: For each constraint $q \sim_{\mathtt{r}x} p$ where q is in $A_{\sigma(\mathtt{y})}$ such that $q \neq q_{\mathtt{y}}$, a new constraint $q' \sim_{\mathtt{r}x} p$ is added, where q' is the version of q in A_k. Likewise, for each constraint $q \sim_{\mathtt{r}x} p$ where p is in $A_{\sigma(\mathtt{y})}$ such that $p \neq q_{\mathtt{y}}$, a new constraint $q \sim_{\mathtt{r}x} p'$ is added (again, p' is the version of p in A_k). Finally, for each constraint of the form $p \sim_{\mathtt{ra}} q_{\mathtt{y}}$, a new constraint $p \sim_{\mathtt{ra}} root(A_k)$ is added.

An example of the splitting step is given in Example 3 below.

In what follows, we assume that $\mathtt{sel}$ is represented by a^i in the sequence $\overline{a} = a^1 \cdots a^m$ so that q_i corresponds to the target of $\mathtt{sel}$. Before performing the actual update, we check whether the operation to be performed tries to dereference a pointer to $\perp$ 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.

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

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

$\mathtt{y\text{->}sel} = \mathtt{x}$ If q_i is a state, then we split $A_{\sigma(\mathtt{y})}$ at the i-th position. Then we put $\overline{\sigma(\mathtt{x})}$ to the i-th position in the right-hand side of the root transition of $A_{\sigma(\mathtt{y})}$; 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 $0 \sim_{\mathtt{r}x} i$ which concerns the removed root reference q_i is then removed from c.

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

$\mathtt{y\text{->}data} \sim \mathtt{x\text{->}data}$ (where $\sim \in \{\prec, \preceq, \succ, \succeq\}$) First, we execute the saturation procedure in order to infer the strongest constraints between $q_\mathtt{y}$ and $q_\mathtt{x}$. Then, if there exists a global constraint $q_\mathtt{y} \sim' q_\mathtt{x}$ that implies $q_\mathtt{y} \sim q_\mathtt{x}$ (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 $q_\mathtt{y} \sim q_\mathtt{x}$ and F_{false} with its negation.

$\mathtt{x} = \mathtt{y}$ or $\mathtt{x} = \mathtt{NULL}$ We simply update σ accordingly.

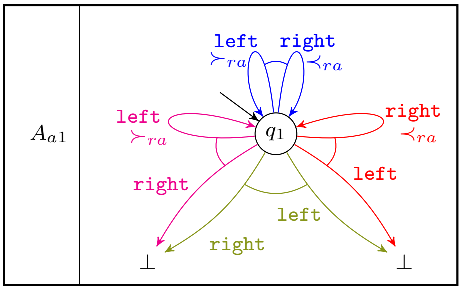
$\mathtt{free}(\mathtt{y})$ First, we split $A_{\sigma(\mathtt{y})}$ at all j-th positions, $1 \leq j \leq m$, that appear in its root transition, then we remove $A_{\sigma(\mathtt{y})}$ from F and set $\sigma(\mathtt{y})$ to undefined. However, to keep all possible data constraints, before removing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

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

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

Example 3 We now present the computation of the abstract configuration that results from executing the program statements which appear at line 9 of the program in Fig. 1 when starting from the abstract configuration described in Fig. 4(a) (for the sake of brevity, we leave out the `newNode` variable and the corresponding TA from the example). In order to compute this abstract configuration, a sequence of two statements including the test statement $\mathtt{x\text{->}right} \neq \mathtt{NULL}$ and the update statement $\mathtt{x} = \mathtt{x\text{->}right}$ is executed. First, the test statement $\mathtt{x\text{->}right} \neq \mathtt{NULL}$ is executed in the following two steps:

1. As can be seen from the FA F_a from Fig. 4(a) encoding BSTs, the root state q_1 of A_{a1} (the only TA of F_a) occurs on the right-hand side of three transitions

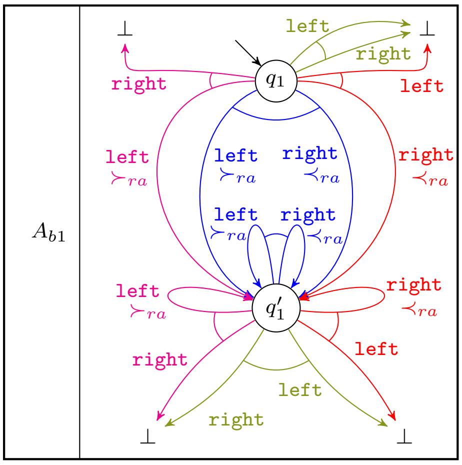


$F_a = \langle A_{a1}, \emptyset \rangle$
$\sigma(\texttt{root}) = 1, \sigma(\texttt{x}) = 1$

A_{a1}

$1a : \underline{q_1} \to \texttt{l}, \texttt{r}(q_1, q_1) : 0 \succ_{\texttt{ra}} 1, 0 \prec_{\texttt{ra}} 2$
$2a : \underline{q_1} \to \texttt{l}, \texttt{r}(\perp, q_1) : 0 \prec_{\texttt{ra}} 2$
$3a : \underline{q_1} \to \texttt{l}, \texttt{r}(q_1, \perp) : 0 \succ_{\texttt{ra}} 1$
$4a : \underline{q_1} \to \texttt{l}, \texttt{r}(\perp, \perp)$

a) An example abstract configuration at line 9 of the program in Fig. 1. The abstract configuration represents a set of BSTs ($\texttt{l}, \texttt{r}$ abbreviates `left`, `right`).



$F_b = \langle A_{b1}, \emptyset \rangle$
$\sigma(\texttt{root}) = 1, \sigma(\texttt{x}) = 1$

A_{b1}

$1b : \underline{q_1} \to \texttt{l}, \texttt{r}(q_1', q_1') : 0 \succ_{\texttt{ra}} 1, 0 \prec_{\texttt{ra}} 2$
$2b : \underline{q_1} \to \texttt{l}, \texttt{r}(\perp, q_1') : 0 \prec_{\texttt{ra}} 2$
$3b : \underline{q_1} \to \texttt{l}, \texttt{r}(q_1', \perp) : 0 \succ_{\texttt{ra}} 1$
$4b : \underline{q_1} \to \texttt{l}, \texttt{r}(\perp, \perp)$
$5b : q_1' \to \texttt{l}, \texttt{r}(q_1', q_1') : 0 \succ_{\texttt{ra}} 1, 0 \prec_{\texttt{ra}} 2$
$6b : q_1' \to \texttt{l}, \texttt{r}(\perp, q_1') : 0 \prec_{\texttt{ra}} 2$
$7b : q_1' \to \texttt{l}, \texttt{r}(q_1', \perp) : 0 \succ_{\texttt{ra}} 1$
$8b : q_1' \to \texttt{l}, \texttt{r}(\perp, \perp)$

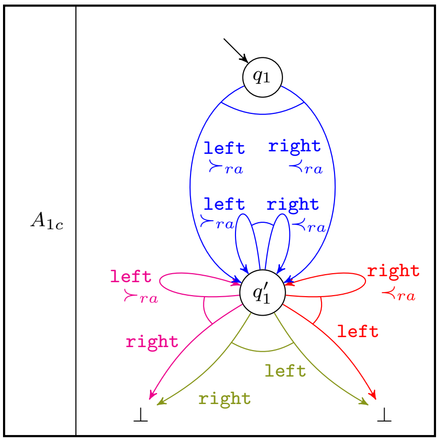
b) An intermediate state of a transformation of the forest automaton F_a from (a) into the form with a single root transition.

Fig. 4 An example of a transformation of an FA into the form with a single root transition.

of A_{a1}, and we will therefore create a state q_1', a copy of q_1, and duplicate to q_1' the four transitions leaving from q_1 (the resulting intermediate FA F_b can be seen in Fig. 4(b)). Then, for each transition $t \in \{1b, 2b, 3b, 4b\}$ leaving from q_1 in A_{1b}, we create a copy of the intermediate FA called F_c, F_d, F_e, and F_f. From the obtained TA A_{1c}, A_{1d}, A_{1e}, and A_{1f}, we subsequently remove all transitions leaving from q_1 other than t, resulting in the four FAs in Fig. 5.

2. The next step is to remove configurations where the root transition of the TA pointed by `x` contains $\perp$ at the the second position of the right-hand side since they do not meet the condition `x->right` $\neq$ `NULL` (they will be processed in the `else` branch). Due to this, the abstract configurations with the FAs F_e and F_f are removed.

Second, the update statement `x` $=$ `x->right` is executed on the abstract configurations shown in Fig. 5(a) and Fig. 5(b). Here, we show the steps only for the abstract configuration from Fig. 5(a), the other one could be computed in a similar manner. The resulting abstract configuration is shown in Fig. 6.

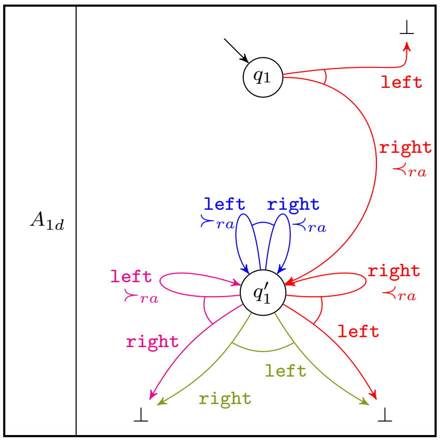


$F_c = \langle A_{1c}, \emptyset\rangle$, $\sigma(\texttt{root}) = 1, \sigma(\texttt{x}) = 1$

A_{1c}

$1c : \underline{q_1} \rightarrow \texttt{1}, \texttt{r}(q'_1, q'_1) : 0 \succ_{\texttt{ra}} 1, 0 \prec_{\texttt{ra}} 2$
$2c : \underline{q'_1} \rightarrow \texttt{1}, \texttt{r}(q'_1, q'_1) : 0 \succ_{\texttt{ra}} 1, 0 \prec_{\texttt{ra}} 2$
$3c : q'_1 \rightarrow \texttt{1}, \texttt{r}(\perp, q'_1) : 0 \prec_{\texttt{ra}} 2$
$4c : q'_1 \rightarrow \texttt{1}, \texttt{r}(q'_1, \perp) : 0 \succ_{\texttt{ra}} 1$
$5c : q'_1 \rightarrow \texttt{1}, \texttt{r}(\perp, \perp)$

a) The resulting FA for the transition $1b : \underline{q_1} \rightarrow \texttt{1}, \texttt{r}(q'_1, q'_1) : 0 \succ_{\texttt{ra}} 1, 0 \prec_{\texttt{ra}} 2$.

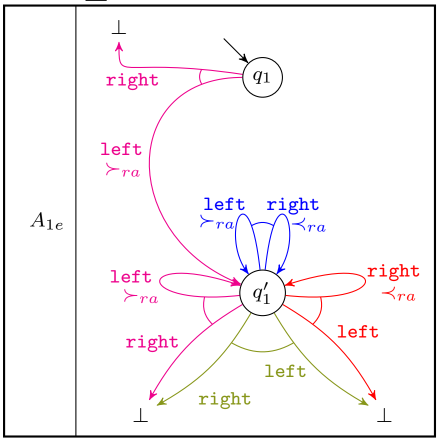


$F_d = \langle A_{1d}, \emptyset\rangle$, $\sigma(\texttt{root}) = 1, \sigma(\texttt{x}) = 1$

A_{1d}

$1d : \underline{q_1} \rightarrow \texttt{1}, \texttt{r}(\perp, q'_1) : 0 \prec_{\texttt{ra}} 2$
$2d : \underline{q'_1} \rightarrow \texttt{1}, \texttt{r}(q'_1, q'_1) : 0 \succ_{\texttt{ra}} 1, 0 \prec_{\texttt{ra}} 2$
$3d : q'_1 \rightarrow \texttt{1}, \texttt{r}(\perp, q'_1) : 0 \prec_{\texttt{ra}} 2$
$4d : q'_1 \rightarrow \texttt{1}, \texttt{r}(q'_1, \perp) : 0 \succ_{\texttt{ra}} 1$
$5d : q'_1 \rightarrow \texttt{1}, \texttt{r}(\perp, \perp)$

b) The resulting FA for the transition $2b : \underline{q_1} \rightarrow \texttt{1}, \texttt{r}(\perp, q'_1) : 0 \prec_{\texttt{ra}} 2$.

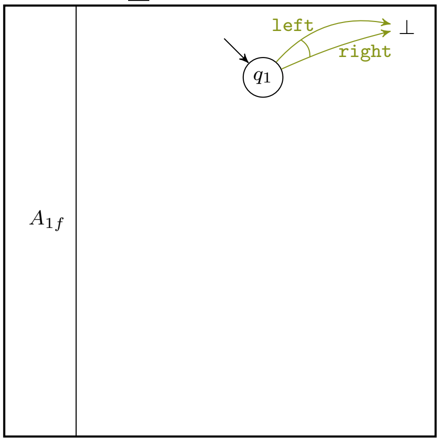


$F_e = \langle A_{1e}, \emptyset\rangle$, $\sigma(\texttt{root}) = 1, \sigma(\texttt{x}) = 1$

A_{1e}

$1e : \underline{q_1} \rightarrow \texttt{1}, \texttt{r}(q'_1, \perp) : 0 \succ_{\texttt{ra}} 1$
$2e : \underline{q'_1} \rightarrow \texttt{1}, \texttt{r}(q'_1, q'_1) : 0 \succ_{\texttt{ra}} 1, 0 \prec_{\texttt{ra}} 2$
$3e : q'_1 \rightarrow \texttt{1}, \texttt{r}(\perp, q'_1) : 0 \prec_{\texttt{ra}} 2$
$4e : q'_1 \rightarrow \texttt{1}, \texttt{r}(q'_1, \perp) : 0 \succ_{\texttt{ra}} 1$
$5e : q'_1 \rightarrow \texttt{1}, \texttt{r}(\perp, \perp)$

c) The resulting FA for the transition $3b : \underline{q_1} \rightarrow \texttt{1}, \texttt{r}(q'_1, \perp) : 0 \succ_{\texttt{ra}} 1$.



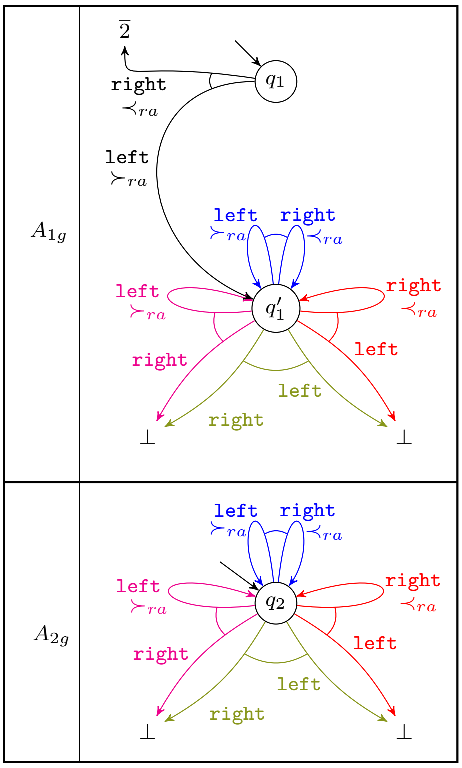
$F_f = \langle A_{1f}, \emptyset\rangle$, $\sigma(\texttt{root}) = 1, \sigma(\texttt{x}) = 1$

A_{1f}

$1f : \underline{q_1} \rightarrow \texttt{1}, \texttt{r}(\perp, \perp)$

d) The resulting FA for the transition $4b : \underline{q_1} \rightarrow \texttt{1}, \texttt{r}(\perp, \perp)$.

Fig. 5 The results of the transformation of the FA from Fig. 4.



$F_g = \langle A_{1g}\, A_{2g}, \{q_1 \prec_{\mathtt{ra}} q_2\}\rangle$
$\sigma(\mathtt{root}) = 1, \sigma(\mathtt{x}) = 2$

A_{1g}

$1g : \underline{q_1} \to \mathtt{l}, \mathtt{r}(q_1', \overline{2}) \;: 0 \succ_{\mathtt{ra}} 1, 0 \prec_{\mathtt{ra}} 2$
$2g : \underline{q_1'} \to \mathtt{l}, \mathtt{r}(q_1', q_1') : 0 \succ_{\mathtt{ra}} 1, 0 \prec_{\mathtt{ra}} 2$
$3g : \underline{q_1'} \to \mathtt{l}, \mathtt{r}(\perp, q_1') \;: 0 \prec_{\mathtt{ra}} 2$
$4g : \underline{q_1'} \to \mathtt{l}, \mathtt{r}(q_1', \perp) \;: 0 \succ_{\mathtt{ra}} 1$
$5g : \underline{q_1'} \to \mathtt{l}, \mathtt{r}(\perp, \perp)$

A_{2g}

$6g : \underline{q_2} \to \mathtt{l}, \mathtt{r}(q_2, q_2) : 0 \succ_{\mathtt{ra}} 1, 0 \prec_{\mathtt{ra}} 2$
$7g : \underline{q_2} \to \mathtt{l}, \mathtt{r}(\perp, q_2) \;: 0 \prec_{\mathtt{ra}} 2$
$8g : \underline{q_2} \to \mathtt{l}, \mathtt{r}(q_2, \perp) \;: 0 \succ_{\mathtt{ra}} 1$
$9g : \underline{q_2} \to \mathtt{l}, \mathtt{r}(\perp, \perp)$

Fig. 6 The FA obtained from F_c from Fig. 5(a) by splitting A_{1c} at the second position.

1. The first step is to compute the new FA resulting from splitting the root transition $1c$ of the TA A_{1c} in the FA F_c in Fig. 5(a) at the second position, yielding the FA F_g. First, we create the TA A_{2g} from A_{1c} by copying it, renaming q_1' to q_2, and making the state q_2 the root state (note that q_1 becomes unreachable in A_{2g}, and so we discard it). Then, we copy A_{1c} to A_{1g} and change the root transition $1c$ of A_{1g} by replacing the state q_1' at the second position of its tuple of children states (corresponding to the selector `right`) by $\overline{2}$ and add the global constraint $q_1 \prec_{\mathtt{ra}} q_2$.
2. The second step is to update the *valuation* σ of both abstract configurations to $\sigma := \sigma\{\mathtt{x} \mapsto 2\}$ meaning that `x` will point to roots of BSTs accepted by A_{2g} whereas $\sigma(\mathtt{root})$ is kept unchanged.

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 TAs so that the roots of trees of accepted forests are cut-points only, which is described in detail below.
3. Finally, we reorder the TAs according to the canonical ordering of cut-points (which are roots of the accepted trees).

Our procedure is an augmentation of that in [15] used to normalize FAs 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, the 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:

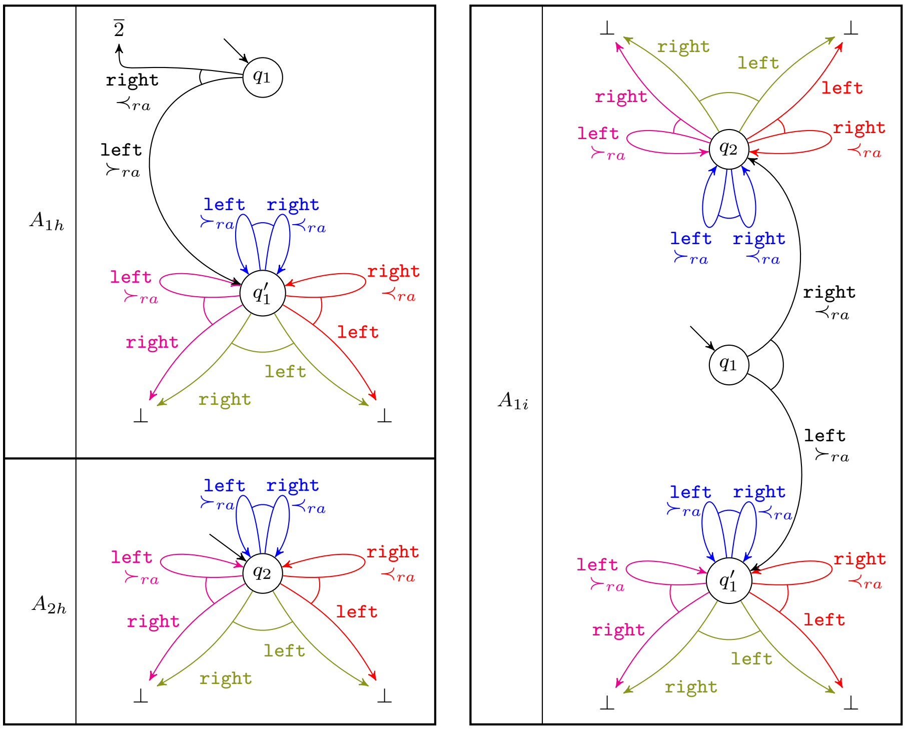
1. all occurrences of $\bar{i}$ among $q_1, \ldots, q_m$ are replaced by q_A, and
2. for all $1 \leq k \leq m$ s.t. q_k can reach $\bar{i}$ by following top-down a sequence of the original rules of Δ_B, the constraint $0 \sim_{\mathtt{ra}} k$ is removed from c unless $q_k \sim_{\mathtt{ra}} q_A \in \varphi$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

Example 4 In this example, we show normalization of the FA in a possible abstract configuration after the execution of line 22 in the program in Fig. 1. The abstract configuration can be seen in Fig. 7(a). Because the roots of the trees accepted by the TA A_{2h} do not correspond to the cut-points of the graphs in $L(F_h)$, we join A_{1h} and A_{2h} in the following way. First, the states and transitions of A_{2h} are copied to A_{1h} and the root state of A_{2h} substitutes the reference $\overline{2}$ in the transition $1h$ of A_{1h}. Afterwards, the TA A_{2h} is removed together with the global data constraint $q_1 \prec_{\mathtt{ra}} q_2$ from the FA. The constraint $0 \prec_{\mathtt{ra}} 2$ is not removed from the root transition $1h$ because $q_1 \prec_{\mathtt{ra}} q_2$ was in the set of global data constraints of F_h before normalization and, therefore, $0 \prec_{\mathtt{ra}} 2$ will still hold. The resulting FA F_i is shown in Fig. 7(b).

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 done using the techniques of [15]. We note that "ordinary FAs" correspond to FAs with no global and no local data constraints. The reduction encodes an FA with data constraints as an FA without data constraints such that its language, when decoded in a particular way, is the same as the language of the original automaton.

An *encoding* of an FA $F = (A_1 \cdots A_n, \varphi)$ with data constraints is an ordinary FA $F^E = (A'_1 \cdots A'_n, \emptyset)$ where the data constraints are written into symbols of transitions. That is, each transition $q \to \overline{a}(q_1, \ldots, q_m) : c$ of $A_i, 1 \leq i \leq n$, is in



$F_h = \langle A_{1h}\, A_{2h}, \{q_1 \prec_{\mathtt{ra}} q_2\}\rangle$
$\sigma(\mathtt{root}) = 1, \sigma(\mathtt{x}) = \bot$

A_{1h}

$1h : \underline{q_1} \to \mathtt{1}, \mathtt{r}(q_1', \overline{2}) \;\; : 0 \succ_{\mathtt{ra}} 1, 0 \prec_{\mathtt{ra}} 2$
$2h : \underline{q_1'} \to \mathtt{1}, \mathtt{r}(q_1', q_1') : 0 \succ_{\mathtt{ra}} 1, 0 \prec_{\mathtt{ra}} 2$
$3h : \underline{q_1'} \to \mathtt{1}, \mathtt{r}(\bot, q_1') \;\; : 0 \prec_{\mathtt{ra}} 2$
$4h : \underline{q_1'} \to \mathtt{1}, \mathtt{r}(q_1', \bot) \;\; : 0 \succ_{\mathtt{ra}} 1$
$5h : \underline{q_1'} \to \mathtt{1}, \mathtt{r}(\bot, \bot)$

A_{2h}

$6h : \underline{q_2} \to \mathtt{1}, \mathtt{r}(q_2, q_2) : 0 \succ_{\mathtt{ra}} 1, 0 \prec_{\mathtt{ra}} 2$
$7h : \underline{q_2} \to \mathtt{1}, \mathtt{r}(\bot, q_2) \;\; : 0 \prec_{\mathtt{ra}} 2$
$8h : \underline{q_2} \to \mathtt{1}, \mathtt{r}(q_2, \bot) \;\; : 0 \succ_{\mathtt{ra}} 1$
$9h : \underline{q_2} \to \mathtt{1}, \mathtt{r}(\bot, \bot)$

a) An abstract configuration.

$F_i = \langle A_{1i}, \emptyset\rangle$
$\sigma(\mathtt{root}) = 1, \sigma(\mathtt{x}) = \bot$

A_{1i}

$1i : \underline{q_1} \to \mathtt{1}, \mathtt{r}(q_1', q_2) : 0 \succ_{\mathtt{ra}} 1, 0 \prec_{\mathtt{ra}} 2$
$2i : \underline{q_1'} \to \mathtt{1}, \mathtt{r}(q_1', q_1') : 0 \succ_{\mathtt{ra}} 1, 0 \prec_{\mathtt{ra}} 2$
$3i : \underline{q_1'} \to \mathtt{1}, \mathtt{r}(\bot, q_1') \;\; : 0 \prec_{\mathtt{ra}} 2$
$4i : \underline{q_1'} \to \mathtt{1}, \mathtt{r}(q_1', \bot) \;\; : 0 \succ_{\mathtt{ra}} 1$
$5i : \underline{q_1'} \to \mathtt{1}, \mathtt{r}(\bot, \bot)$
$6i : q_2 \to \mathtt{1}, \mathtt{r}(q_2, q_2) : 0 \succ_{\mathtt{ra}} 1, 0 \prec_{\mathtt{ra}} 2$
$7i : q_2 \to \mathtt{1}, \mathtt{r}(\bot, q_2) \;\; : 0 \prec_{\mathtt{ra}} 2$
$8i : q_2 \to \mathtt{1}, \mathtt{r}(q_2, \bot) \;\; : 0 \succ_{\mathtt{ra}} 1$
$9i : q_2 \to \mathtt{1}, \mathtt{r}(\bot, \bot)$

b) The abstract configuration from (a) after normalization.

Fig. 7 An example of running normalization on the abstract configuration obtained from the program in Fig. 1 after executing line 22.

A_i' replaced by the transition $q \to \langle (a_1, \ell_1, g) \cdots (a_m, \ell_m, g)\rangle (q_1, \ldots, q_m) : \emptyset$ where for $1 \le j \le m$, ℓ_j is the subset of c containing the local constraints involving j, and g encodes the global constraints involving q as follows: Let r be the root state of some $A_k, 1 \le k \le n$, that does not appear within any right-hand side of a rule. Then for a global constraint $q \sim_{\mathtt{rx}} r$, or $r \sim_{\mathtt{rx}} q$, g contains $0 \sim_{\mathtt{rx}} k$, or

$k \sim_{\mathtt{rx}} 0$, respectively. The language of A'_i 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 $j \sim_{\mathtt{rx}} k$ for $j, k \in \mathbb{N}_0$.

To show that testing inclusion of encoded FAs is a sound approximation of language inclusion test of FAs with constraints, we need to establish a correspondence between languages of the encoded FAs and languages of the original ones. For this, we define a *decoding* of a forest $t'_1 \cdots t'_n$ from a language of an encoded FA over Γ^E as the set of forests $t_1 \cdots t_n$ over Γ such that $t_1 \cdots t_n$ arises 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 $\ell, g \subseteq \mathbb{C}$, we have $(a, \ell, g)_{t'_i}(u) = v$ iff: (1) $a_{t_i}(u) = v$ and (2) for all $1 \leq j \leq n$: if $0 \sim_{\mathtt{rx}} j \in \ell$, then $u \sim_{\mathtt{rx}} v$ (in t_i), and if $0 \sim_{\mathtt{rx}} j \in g$, then $u \sim_{\mathtt{rx}} \mathit{root}(t_j)$ (symmetrically for $j \sim_{\mathtt{rx}} 0$). The notion of decoding allows us to summarise the correspondence of languages of FAs and languages of their encodings as follows.

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

Proof Let $F = \langle A_1 \cdots A_n, \varphi \rangle$ and $F^E = \langle A'_1 \cdots A'_n, \emptyset \rangle$. We first prove that every forest $t_1 \cdots t_n$ accepted by F is a 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 $A'_1 \ldots A'_n$ on the forest $t'_1 \cdots t'_n$ of which $t_1 \cdots t_n$ is a decoding such that for every ρ_i, $1 \leq i \leq n$, we will construct the run ρ'_i. Let us first simplify the notation by denoting ρ_i, t_i, ρ'_i, t'_i, A_i, and A'_i by ρ, t, ρ', t', A, and A', respectively. The run ρ' is constructed as follows. $V_{t'} = V_t$ and $\lambda_{t'}$ can be chosen arbitrarily. For every $v \in V_t$ such that $a^1_t(v) = v_1, \ldots, a^m_t(v) = v_m$ are the 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 $v, v_1, \ldots, v_m$ in t, and also global constraints $q \sim_{\mathtt{rx}} r, r \sim_{\mathtt{rx}} q \in \varphi$ are satisfied by v and $\rho_k(r)$ for the k such that r is a state of A_k. (by the definition of a run). The 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 the encoding of δ ($\bar{\alpha} = \langle (a_1, \ell_1, g) \cdots (a_m, \ell_m, g) \rangle$ where g contains encoded the part of φ involving q and $c = \ell_1 \cup \cdots \cup \ell_m$). ρ' is obviously a run of A'. The described 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 children of v, a pair of transitions (δ, δ') of A and A', respectively, where δ and δ' are the rules used within ρ and ρ', respectively, to label $v, v_1, \ldots, v_m$.

Let us argue that $t_1 \cdots t_n$ is indeed a decoding of $t'_1 \cdots t'_n$. It is trivially satisfied for all $1 \leq i \leq n$ that $V_{t_i} = V_{t'_i}$ and that every node has the same children in both forests. In order to argue that data values in $t_1 \cdots t_n$ satisfy the constraints encoded in $t'_1 \cdots t'_n$ as required by the definition of decoding, we let $v \in V_{t_i}$ be a node with children $v_1, \ldots, v_m$ such that $f(v, v_1, \ldots, v_m) = (\delta, \delta')$ where $\delta = q \to \overline{a}(q_1, \ldots, q_m) : c$ and $\delta' = q \to \bar{\alpha}(q_1, \ldots, q_m) : \emptyset$ and $\bar{\alpha} = \langle (a_1, \ell_1, g) \cdots (a_m, \ell_m, g) \rangle$. Then the constraints imposed on the data value of v within $t_1 \cdots t_n$ by φ and those imposed by c due to the use of δ are the same as the constraints enforced on v due to $\bar{\alpha}$ when $t'_1 \cdots t'_n$ is decoded into $t_1 \cdots t_n$. In detail, c contains a local constraint $0 \sim k$ iff ℓ_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. Secondly, there is a global constraint of the form $q \sim r \in \varphi$ such that r is the root state of A_k (not appearing within right-hand sides of its transitions) iff $0 \sim k \in g$ (and analogically for the symmetrical cases). In the run of A, $q \sim r$ enforces that $v \sim u$ where u is the root of t_k. Notice that u cannot be any other node than the root since r does not appear within right-hand sides of transitions of A_k. $v \sim u$ is precisely what is enforced due to $0 \sim k \in g$ when decoding $t'_1 \cdots t'_n$.

Secondly, we prove that every decoding $t_1 \cdots t_n$ of a forest $t'_1 \cdots t'_n \in L(F^E)$ is accepted by F. We will do that by showing that every n-tuple of runs $\rho'_1, \dots, \rho'_n$ of $A'_1, \dots, A'_n$ on $t_1, \dots, t_n$ respectively also encodes runs of $A_1, \dots, A_n$ on $t_1, \dots, t_n$ respectively.

Recall first that by the definition of a decoding, for each $1 \leq i \leq n$, t_i and t'_i have the same sets of nodes and every node have the same tuple of children. To simplify the notation, let t, ρ', t', A, A' be denoted as $t_i, \rho'_i, t'_i, A_i, A'_i$ respectively. Let $v \in V_{t'}$ and let $\alpha^1_{t'}(v) = v_1, \dots, \alpha^m_{t'}(v) = v_m$ be the edges of t' with the source v where for all $1 \leq j \leq m$, $\alpha_j = (a_j, \ell_j, g)$. By the definition of a decoding, v satisfies all constraints encoded within $\bar{\alpha}$. Since t' is accepted by A', there is a transition of A' of the form $\delta' = q \to \bar{\alpha}(q_1, \dots, q_m) : \emptyset$ such that $\rho'(v) = q$, $\rho'(v_1) = q_1, \dots, \rho'(v_m) = q_m$. By the definition of encoding, δ' was created from a rule $\delta = q \to \overline{a}(q_1, \dots, q_m) : c$ of A where $\ell_1 \cup \cdots \cup \ell_m = c$ and g encodes all global constraints involving q and a root state r which does not appear within a right-hand side of any rule. These constraints are precisely those encoded within $\bar{\alpha}$ and hence required to hold for v in $t_1 \cdots t_n$ by decoding. ρ' is thus indeed a run of A since for every v and its children $v_1, \dots, v_m$, there is a rule δ which can be used according to the definition of a run. $\square$

A direct consequence of Lemma 1 is that if $L(F^E_A) \subseteq L(F^E_B)$, then $L(F_A) \subseteq L(F_B)$. We can thus use the language inclusion checking procedure of [15] for ordinary FAs to safely approximate language inclusion of FAs with data constraints.

This language inclusion test is not complete, the above implication does not hold in the opposite direction. There are two reasons for this. First, encoding translates a constraint of F_B that is strictly weaker than a constraint of F_A into two different and unrelated labels. This may result in the situation that even though $L(F_A) \subseteq L(F_B)$, language inclusion of encodings of FAs does not hold due to the reason that the trees accepted are labelled by different symbols. For instance, let $F_A = (A_1, \emptyset)$ where A_1 contains only a single transition $\delta_A = q \to a(\overline{1}) : \{0 \prec_{\mathtt{rr}} 1\}$ and $F_B = (B_1, \emptyset)$ where B_1 also contains only a single transition $\delta_B = r \to a(\overline{1}) : \emptyset$. It holds that $L(F_A) \subseteq L(F_B)$ (indeed, $L(F_A) = \emptyset$ due to the strict inequality on the root), but $L(F^E_A)$ is incomparable with $L(F^E_B)$. 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 $0 \prec_{\mathtt{rr}} 1$ plays no role. The second source of incompleteness of the inclusion test is that decodings of some forests accepted by F^E_A and F^E_B may be empty due to inconsistent data constraints. If the set of such inconsistent forests of F^E_A is not included in that of F^E_B, then $L(F^E_A)$ cannot be included in $L(F^E_B)$, 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 problem of inconsistent data constraints since it does not seem to occur in practice, as witnessed by our experiments. On the other hand, the issue of incompatible encodings of related data constraints appears to be of a practical consequence. We address it with a quite simple transformation of F^E_B: we pump-up the TAs of F^E_B by variants of their transitions which encode stronger data constraints than originals and match the data constraints on transitions of F_A^E. Since we are adding transitions with stronger constraints than the existing ones, this does not change the language of F_B. For instance, in our previous example, we add the transition $r \to a(\overline{1}) : \{0 \prec_{\mathtt{rr}} 1\}$ to B_1. This transition, when encoded, can then correspond to the encoded version of the transition $q \to a(\overline{1}) : \{0 \prec_{\mathtt{rr}} 1\}$ of A_1 and the language inclusion of the encodings will hold.

Formally, we call a sequence $\overline{\alpha} = \langle(a_1, \ell_1, g) \cdots (a_m, \ell_m, g)\rangle \in (\Gamma^E)^m$ *stronger* than a sequence $\overline{\beta} = \langle(a_1, \ell_1', g') \cdots (a_m, \ell_m', g')\rangle$ iff $\bigwedge g \implies \bigwedge g'$ and for all $1 \leq i \leq m$, $\bigwedge \ell_i \implies \bigwedge \ell_i'$. 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 TAs 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). The FA $F_B^{E^+}$ can thus be used within language inclusion checking in the place of F_B^E. This technique prevents the inclusion check to fail because of incompatible encodings of data constraints. Its soundness is summarised by the following lemma.

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

Proof (sketch) Since the transformation from F_2^E to $F_2^{E^+}$ adds only versions of existing rules encoding stronger constraints, the sets of decodings of forest of $F_2^{E^+}$ is the same the set of decodings of forests of F_2^E. The statement then follows immediately from Lemma 1. $\square$

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 [15] 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 [15] (we use the one from Section 4, used to define canonical forests).

6 Boxes

Forest automata, as defined in Section 4, can represent graphs with cut-points of an unbounded in-degree as, e.g., in SLLs with head/tail pointers (indeed there can be any number of references from leaf nodes to a certain root). However, the basic definition of FAs cannot deal with 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 [15] is to allow FAs to use other nested FAs, called *boxes*, as symbols to "hide" recurring subgraphs and in this way eliminate cut-points. The alphabet of a box itself may also include boxes, however, these boxes are required to form a hierarchy, they cannot be recursively nested. To make the semantics of a box clear, we will need to extend the definitions of an FA from Section 4 to allow so-called ports. Ports are nodes of a graph hidden within a box at which should be the hidden graph connected to its surroundings. For simplicity of presentation, we give only a simplified version of the definition in [15], 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* $\Box$ over $\langle \Gamma, \Omega \rangle$, where Γ does not contain $\Box$, 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 $g \rightsquigarrow g'$) 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 $\rightsquigarrow^*$ to denote the reflexive transitive closure of $\rightsquigarrow$. 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 $g \rightsquigarrow^* g'$.

In a verification run, boxes are automatically inferred using the techniques presented in [17]. 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.

Extension of forest automata of [15,17] by data constraints must be reflected within treatment of boxes. Particularly, in order not to lose information stored within data constraints, folding and unfolding require 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 TAs within the box which correspond to its input and output port.

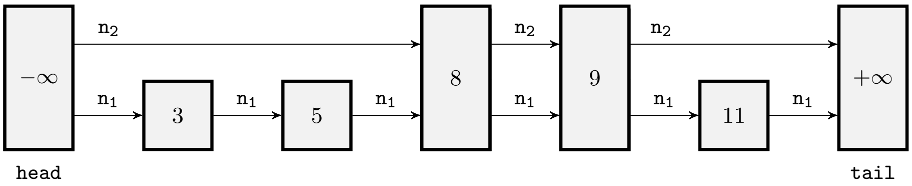


Fig. 8 An example of a 2-level skip list.

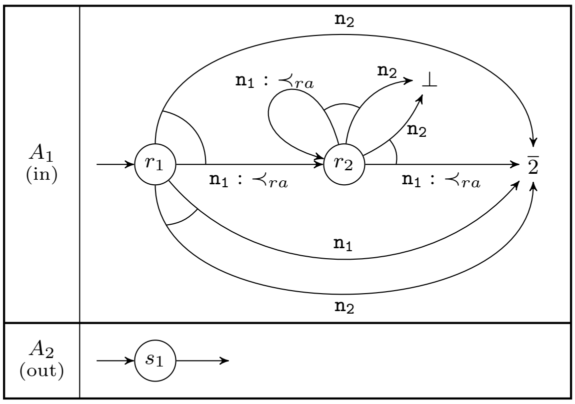
Example 5 In this example we show how to unfold and fold boxes on a sample abstract configuration of a program manipulating a 2-level skip list. A skip list is a linked list sorted by keys. Each node is assigned a height, either 1 or 2, and one successor for every level. For example, a node of level 2 has two next pointers, here called $\mathtt{n}_1$ and $\mathtt{n}_2$, where $\mathtt{n}_1$ points to the next node of level 1 and $\mathtt{n}_2$ points to the next node of level 2. Fig. 8 shows an example configuration of a 2-level skip list with integer keys (the nodes `head` and `tail` with the keys $-\infty$ and $+\infty$ respectively are used as sentinels).

We can see from Fig. 8 that each internal node of level 2 is a cut-point. In order to be able to represent a skip list of any length, it is necessary to introduce a box that effectively hides these cut-points. We use, in particular, the box $\mathtt{skl}_2$ from Fig. 9(a), which represents all skip list segments between a pair of nodes of level 2. Fig. 9(b) shows an abstract configuration of a skip list with 3 nodes of level 2: the `head` node, the `tail` node, and one regular node in between. The number of level 1 nodes (hidden inside the two `skl2` boxes) is arbitrary. Note that the output port of `skl2` contains an automaton accepting ε; this is because there are no transitions leading from the output port of the box.

Fig. 9(c) shows an *unfolding* of the first occurrence of the $\mathtt{skl}_2$ box in the FA. Intuitively, the unfolding proceeded in the following steps:

1. As a preparatory step for replacing the use of $\mathtt{skl}_2$ on the transition $1b$ by the contents of the box represented by $\mathtt{skl}_2$, the TA B_1 was split at the state t_2 to isolate the transition $1b$. This produced two auxiliary TAs B_1' and B_3' consisting of the transitions $1b' : t_1 \to \mathtt{skl}_2(\overline{3}) : 0 \prec_{ra} 1$ and $2b : t_2 \to \mathtt{skl}_2(\overline{2}) : 0 \prec_{ra} 1$, respectively, with $\overline{3}$ being a newly introduced cut-point.
2. Subsequently, the TA A_1 corresponding to the input port of $\mathtt{skl}_2$ was inserted in between of t_1 and $\overline{3}$ instead of the transition $1b'$ over $\mathtt{skl}_2$, yielding the TA B_1''. (Notice that if the transition $1b'$ led—via other symbols than $\mathtt{skl}_2$—to more targets than just $\overline{3}$, the part of $1b'$ leading from t_1 to such targets would be preserved and merged with the root transitions of A_1.) On the other hand, the TA A_2 corresponding to the output port of $\mathtt{skl}_2$ was merged with the transtion $2b$ leading from t_2. However, since A_2 accepts ε, the resulting transition $6c$ of B_3'' remains the same as the original transition $2b$. (The TA B_2 was copied into the TA B_2'' without any modification.)
3. The local data constraint from the transition $1b : t_1 \to \mathtt{skl}_2(t_2) : 0 \prec_{ra} 1$ was transformed into the global data constraint $t_1 \prec_{ra} t_2$ during the unfolding.

The subsequent saturation then also generated the local constraints $0 \prec_{ra} 1$ and $0 \prec_{ra} 2$ on the transitions $1c$ and $2c$ from t_1 to $\overline{3}$, and the global constraints $r_2 \prec_{ra} t_2$ and $r_2 \prec_{ra} u_1$ (these changes are emphasized by a bold typeface in Fig. 9(c)).



$\mathtt{skl}_2 = \langle F_{\mathtt{skl}_2}, 1, 2 \rangle$

$F_{\mathtt{skl}_2} = \langle A_1\, A_2, \emptyset \rangle$

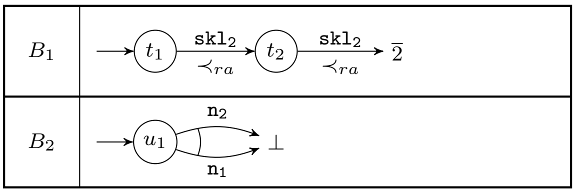
A_1

$1a : \underline{r_1} \to \mathtt{n}_1, \mathtt{n}_2(r_2, \overline{2}) \ : 0 \prec_{\mathtt{ra}} 1$
$2a : \underline{r_1} \to \mathtt{n}_1, \mathtt{n}_2(\overline{2}, \overline{2})$
$3a : r_2 \to \mathtt{n}_1, \mathtt{n}_2(r_2, \bot) : 0 \prec_{\mathtt{ra}} 1$
$4a : r_2 \to \mathtt{n}_1, \mathtt{n}_2(\overline{2}, \bot) \ : 0 \prec_{\mathtt{ra}} 1$

A_2

$5a : \underline{s_1} \to \varepsilon()$

a) The 2-level skip list box $\mathtt{skl}_2$.



$F = \langle B_1\, B_2, \varphi \rangle$
$\varphi = \{t_1 \prec_{\mathtt{ra}} u_1, t_2 \prec_{\mathtt{ra}} u_1\}$
$\sigma(\mathtt{head}) = 1, \sigma(\mathtt{tail}) = 2$

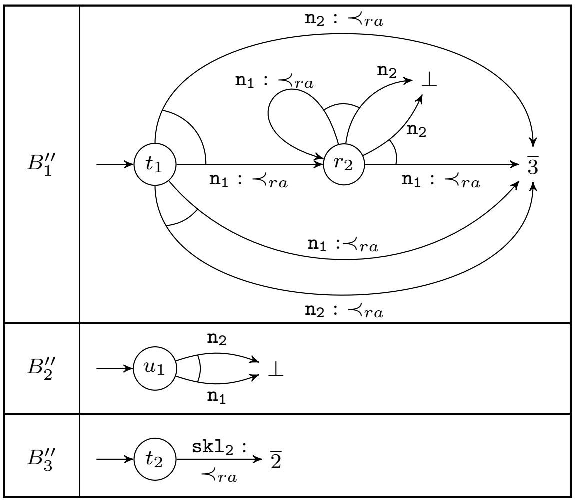
B_1

$1b : \underline{t_1} \to \mathtt{skl}_2(t_2) \quad : 0 \prec_{\mathtt{ra}} 1$
$2b : t_2 \to \mathtt{skl}_2(\overline{2}) \quad : 0 \prec_{\mathtt{ra}} 1$

B_2

$3b : \underline{u_1} \to \mathtt{n}_1, \mathtt{n}_2(\bot, \bot)$

b) A heap containing a skip list with two segments.



$F' = \langle B''_1\, B''_2\, B''_3, \varphi' \rangle$
$\varphi' = \{t_1 \prec_{\mathtt{ra}} u_1, t_2 \prec_{\mathtt{ra}} u_1,$
$\mathbf{t_1} \prec_{\mathtt{ra}} \mathbf{t_2}, \mathbf{r_2} \prec_{\mathtt{ra}} \mathbf{u_1},$
$\mathbf{r_2} \prec_{\mathtt{ra}} \mathbf{t_2}\}$
$\sigma(\mathtt{head}) = 1, \sigma(\mathtt{tail}) = 2$

B''_1

$1c : \underline{t_1} \to \mathtt{n}_1, \mathtt{n}_2(r_2, \overline{3}) \ : \ 0 \prec_{\mathtt{ra}} 1, \mathbf{0} \prec_{\mathtt{ra}} \mathbf{2}$
$2c : \underline{t_1} \to \mathtt{n}_1, \mathtt{n}_2(\overline{3}, \overline{3}) \ : \ \mathbf{0} \prec_{\mathtt{ra}} \mathbf{1}, \mathbf{0} \prec_{\mathtt{ra}} \mathbf{2}$
$3c : r_2 \to \mathtt{n}_1, \mathtt{n}_2(r_2, \bot) : 0 \prec_{\mathtt{ra}} 1$
$4c : r_2 \to \mathtt{n}_1, \mathtt{n}_2(\overline{3}, \bot) \ : 0 \prec_{\mathtt{ra}} 1$

B''_2

$5c : \underline{u_1} \to \mathtt{n}_1, \mathtt{n}_2(\bot, \bot)$

B''_3

$6c : \underline{t_2} \to \mathtt{skl}_2(\overline{2}) \quad : 0 \prec_{\mathtt{ra}} 1$

c) Unfolding of the first occurrence of the $\mathtt{skl}_2$ box in (b).

Fig. 9 An example of unfolding of a box representing a 2-level skip list segment. For the sake of conciseness, we omitted all $\prec_{\mathtt{rr}}$ constraints which are subsumed by $\prec_{\mathtt{ra}}$ constraints.

The inverse operation of *folding* would transform the FA from Fig. 9(c), while using the $\mathtt{skl}_2$ box, into the FA in Fig. 9(b). See [15] for more details on box folding and unfolding.

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 [17] which was modified to remove the dependency of the algorithm on sortedness). Although the examples are of a smaller size, they are very challenging as they include complex manipulation with dynamic memory that may depend on data values stored in memory cells.

Table 4 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. The experiments were run on a machine with the Intel Core i5-480M @2.67 GHz CPU and 5 GiB of RAM.

Compared with works [20, 27, 6, 24], 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.

7.1 Discussion

In the above, we described evaluation of our approach on programs manipulating skip lists of two and three levels. A natural question would be why we limit ourselves to two and three levels and not consider skip lists of even higher or, which would be the best case, of an arbitrary level.

Based on our experience, already going from 2-level to 3-level skip lists makes a huge difference in difficulty, due to the occurrence of a combinatorial explosion in the number of shapes considered by our approach. In order to make handling

Table 4 Results of the experiments.

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

Example		Time [s]
SL_2	insert	9.65
	delete	10.14
SL_3	insert	56.99
	delete	57.35
BST	insert	6.87
	delete	15.00
	left rotate	7.35
	right rotate	6.25

of a 3-level skip list feasible, we had to refine our finite height abstraction from a quite coarse one, which was sufficient for the other considered data structures, to take into account the number of unique paths from a state to a root reference (this step is described in more detail in Section 5 of [17] for the case without data relations). For the case of 4-level skip lists, this ad-hoc abstraction refinement was not sufficient and our experiments did not finish in reasonable time.

Moreover, in order to support skip lists with an arbitrary number of next selectors, these would need to be stored in a dynamic list, therefore making the data structure yet more complex. Even more, the support of a data structure of an arbitrary level in the current technique would need to use recursive nesting of boxes, which is not supported. Allowing this would demand to rewrite the box learning algorithm to be able to find such recursive boxes, and the operations for manipulating those, including the language inclusion algorithm. These modifications are quite challenging and an interesting future research direction.

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 FAs 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.

Acknowledgment. We thank the anonymous reviewers for their useful feedback on the paper.

References

1. Parosh Aziz Abdulla, Muhsin Atto, Jonathan Cederberg, and Ran Ji. Automated analysis of data-dependent programs with dynamic memory. In *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis—ATVA'09*, volume 5799 of *LNCS*, pages 197–212. Springer, 2009.
2. Parosh Aziz Abdulla, Frédéric Haziza, Lukáš Holík, Bengt Jonsson, and Ahmed Rezine. An integrated specification and verification technique for highly concurrent data structures. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems—TACAS'13*, volume 7795 of *LNCS*, pages 324–338. Springer, 2013.
3. Parosh Aziz Abdulla, Lukáš Holík, Bengt Jonsson, Ondřej Lengál, Cong Quy Trinh, and Tomáš Vojnar. Verification of heap manipulating programs with ordered data by extended forest automata. In *Proceedings of the 11th International Symposium on Automated Technology for Verification and Analysis—ATVA'13*, volume 8172 of *LNCS*, pages 224–239. Springer, 2013.
4. Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Lazy shape analysis. In *Proceedings of the 18th International Conference on Computer Aided Verification—CAV'06*, volume 4144 of *LNCS*, pages 532–546. Springer, 2006.
5. Jesse Bingham and Zvonimir Rakamarić. A logic and decision procedure for predicate abstraction of heap-manipulating programs. In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation—VMCAI'06*, volume 3855 of *LNCS*, pages 207–221. Springer, 2006.
6. Ahmed Bouajjani, Marius Bozga, Peter Habermehl, Radu Iosif, Pierre Moro, and Tomáš Vojnar. Programs with lists are counter automata. *Formal Methods in System Design*, 38(2):158–192, 2011.
7. Ahmed Bouajjani, Cezara Drăgoi, Constantin Enea, and Mihaela Sighireanu. Accurate invariant checking for programs manipulating lists and arrays with infinite data. In *Proceedings of the 10th International Symposium on Automated Technology for Verification and Analysis—ATVA'12*, volume 7561 of *LNCS*, pages 167–182. Springer, 2012.
8. Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomáš Vojnar. Abstract regular (tree) model checking. *International Journal on Software Tools for Technology Transfer*, 14(2):167–191, 2012.
9. Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *Proceedings of the 35th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages—POPL'08*, pages 247–260. ACM, 2008.
10. Bor-Yuh Evan Chang, Xavier Rival, and George C. Necula. Shape analysis with structural invariant checkers. In *Proceedings of the 14th International Static Analysis Symposium—SAS'07*, volume 4634 of *LNCS*, pages 384–401. Springer, 2007.
11. Shaunak Chatterjee, Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakaramarić. A reachability predicate for analyzing low-level software. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems—TACAS'07*, volume 4424 of *LNCS*, pages 19–33. Springer, 2007.
12. Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao 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.
13. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages—POPL'77*, pages 238–252. ACM, 1977.
14. Kamil Dudka, Petr Peringer, and Tomáš Vojnar. Byte-precise verification of low-level list manipulation. In *Proceedings of the 20th International Static Analysis Symposium—SAS'13*, volume 7935 of *LNCS*, pages 215–237. Springer, 2013.
15. Peter Habermehl, Lukáš Holík, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. Forest automata for verification of heap manipulation. *Formal Methods in System Design*, 41(1):83–106, 2012.

16. Jonathan Heinen, Thomas Noll, and Stefan Rieger. Juggrnaut: Graph grammar abstraction for unbounded heap structures. In *Proceedings of the 3rd International Workshop on Harnessing Theories for Tool Support in Software—TTSS'09*, volume 266 of *ENTCS*, pages 93–107. Elsevier, 2010.
17. Lukáš Holík, Ondřej Lengál, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. Fully automated shape analysis based on forest automata. In *Proceedings of the 25th International Conference on Computer Aided Verification—CAV'13*, volume 8044 of *LNCS*, pages 740–755. Springer, 2013.
18. Jakob L. Jensen, Michael E. Jørgensen, Michael I. Schwartzbach, and Nils Klarlund. Automatic verification of pointer programs using monadic second-order logic. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation—PLDI'97*, pages 226–234. ACM, 1997.
19. Shuvendu K. Lahiri and Shaz Qadeer. Back to the future: Revisiting precise program verification using SMT solvers. In *Proceedings of the 35th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages—POPL'08*, pages 171–182. ACM, 2008.
20. Alexey Loginov, Thomas Reps, and Mooly Sagiv. Abstraction refinement via inductive learning. In *Proceedings of the 17th International Conference on Computer Aided Verification—CAV'05*, volume 3576 of *LNCS*, pages 519–533. Springer, 2005.
21. Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. Automatic numeric abstractions for heap-manipulating programs. In *Proceedings of the 37th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages—POPL'10*, pages 211–222. ACM, 2010.
22. Scott McPeak and George C. Necula. Data structure specifications via local equality axioms. In *Proceedings of the 17th International Conference on Computer Aided Verification—CAV'05*, volume 3576 of *LNCS*, pages 476–490. Springer, 2005.
23. William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, June 1990.
24. Shengchao Qin, Guanhua He, Chenguang Luo, Wei-Ngan Chin, and Xin Chen. Loop invariant synthesis in a combined abstract domain. *Journal of Symbolic Computation*, 50:386–408, 2013.
25. Zvonimir Rakamarić, Roberto Bruttomesso, Alan J. Hu, and Alessandro Cimatti. Verifying heap-manipulating programs in an SMT framework. In *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis—ATVA'07*, volume 4762 of *LNCS*, pages 237–252. Springer, 2007.
26. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.
27. Thomas. Wies, Viktor Kuncak, Karen Zee, Andreas Podelski, and Martin Rinard. On verifying complex properties using symbolic shape analysis. In *Proceedings of Heap Analysis and Verification 2007—HAV'07*, 2007.
28. Thomas Wies and Andreas Podelski. Counterexample-guided focus. In *Proceedings of the 37th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages—POPL'10*, pages 249–260. ACM, 2010.
29. Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In *Proceedings of the 20th International Conference on Computer Aided Verification—CAV'08*, volume 5123 of *LNCS*, pages 385–398. Springer, 2008.
30. Karen Zee, Viktor Kuncak, and Martin C. Rinard. Full functional verification of linked data structures. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation—PLDI'08*, pages 349–361. ACM, 2008.