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

Peter Habermehl · **Radu Iosif** · **Tomáš Vojnar**

Automata-based Verification of Programs with Tree Updates

the date of receipt and acceptance should be inserted later

Abstract This paper describes a verification framework for Hoare-style pre- and post-conditions of programs manipulating balanced tree-like data structures. Since the considered verification problem is undecidable, we appeal to the standard semi-algorithmic approach in which the user has to provide loop invariants, which are then automatically checked, together with the program pre- and post-conditions. We specify sets of program states, representing tree-like memory configurations, using *Tree Automata with Size Constraints* (TASC). The main advantage of this new class of tree automata is that they recognise tree languages based on arithmetic reasoning about the lengths (depths) of various (possibly all) paths in trees, like, e.g., in *AVL trees* or *red-black trees*. TASCs are closed under union, intersection, and complement, and their emptiness problem is decidable. Thus we obtain a class of automata which are an interesting theoretical contribution by itself. Further, we show that, under few restrictions, one can automatically compute the effect of tree-updating program statements on the set of configurations represented by a TASC, which makes TASC a practical verification tool. We tried out our approach on the insertion procedure for red-black trees, for which we verified that the output on an arbitrary balanced red-black tree is also a balanced red-black tree.

A short version of this paper appeared in the Proceedings of TACAS 2006. The work was supported by the French Ministry of Research (RNTL project AVERILES), the Czech Science Foundation within the project 102/07/0322, the Czech-French Barrande project MEB 020840, and the Czech Ministry of Education by the project MSM 0021630528.

P. Habermehl
LIAFA, Université Paris Diderot—Paris 7/CNRS, Case 7014, F-75205 Paris 13, France,
E-mail: haberm@liafa.jussieu.fr

R. Iosif
VERIMAG, Université Joseph Fourier/CNRS/INPG, 2 av. de Vignate, F-38610 Gières, France,
E-mail: iosif@imag.fr

T. Vojnar
FIT, Brno University of Technology, Božetěchova 2, CZ-61266, Brno, Czech Republic,
E-mail: vojnar@fit.vutbr.cz

1 Introduction

Verification of programs using dynamic memory primitives, such as allocation, deallocation, and pointer manipulations, is crucial for a feasible method of software verification. In this paper, we address the problem of proving correctness of programs that manipulate balanced tree-like data structures. Such structures are very often applied to implement in an efficient way lookup tables, associative arrays, sets, or similar higher-level structures, especially when they are used in critical applications like real-time systems, kernels of operating systems, etc. Therefore, a number of such search tree structures like the AVL trees, red-black trees, splay trees, and so on [11] have been introduced.

Tree automata [9] are a powerful formalism for specifying and reasoning about infinite sets of trees. However, there are two major obstacles against the broad use of tree automata in program verification:

- Imperative programs perform destructive updates of selector fields, changing a tree-shaped data structure by temporarily introducing sharing of branches and/or loops. For instance, this is the case of *tree rotations* [23] which are implemented as a finite sequence of selector updates introducing a loop in the tree in order to re-establish the tree-like shape later on.
- Tree automata represent regular sets of trees, which is not sufficient when one needs to reason in terms of *balanced* trees as in the case of AVL and red-black tree algorithms.

In order to overcome the first problem, we observe that most algorithms working on balanced trees [11] use *tree rotations* and *addition/removal of leaf nodes* to/from a tree as the only operations that change the structure of the input tree. In our framework, we consider these updates as single (atomic) steps in the program. The correctness of their implementation, using lower-level pointer operations, can, however, be checked separately in a different formalism such as, for example, 3-valued predicate logic with transitive closure [24], or tree automata extended with additional "routing" expressions on the tree backbone as in [17] or in [5], where the so-called abstract regular tree model checking is used.

The second inconvenience is solved in the present paper by introducing a novel class of tree automata, called *Tree Automata with Size Constraints* (TASC). TASC are tree automata whose actions are triggered by arithmetic constraints involving the *sizes* of the subtrees at the current node. The size of a tree is a numerical function defined inductively on the tree structure such as, for instance, the height, the maximum number of black nodes on all paths, etc. The main advantage of using TASC in program verification is that they recognise non-regular sets of tree languages, such as the *AVL trees*, the *red-black trees*, and, in general, sets of trees involving arithmetic reasoning about the lengths (depths) of various (possibly all) paths in the trees. We show that the class of TASC is closed under the operations of union, intersection, and complement. Also, the emptiness problem is decidable. We thus obtain a class of automata which are a significant theoretical contribution by itself. Moreover, the semantics of the programs performing tree updates (node recolouring, rotations, leaf nodes appending/removal) can be effectively represented as changes on the structure of the automata.

In our verification approach based on TASC, the user has to provide the precondition and postcondition of the (sequential) imperative program being verified

as well as loop invariants for all loops present in the program. The verification problem then consists in checking validity of Hoare triples of the form $\{P\}C\{Q\}$, where P and Q are TASC-encoded sets of configurations corresponding to the precondition or postcondition of the program or to some loop invariant, and C is a loop-free fragment of the program to be verified. Next, we reduce this verification problem to the TASC language emptiness problem. Note that while the pre- and postconditions and loop invariants are to be specified by the user, checking the validity of the verification conditions is *fully automated* and *exact* in our framework.

We tested our approach on an example of the insertion algorithm for the red-black trees, for which we verify that for a balanced red-black tree input, the output of the insertion algorithm is also a balanced red-black tree, i.e., the number of black nodes is the same on each path.

Related Work. Sound verification of complex properties of programs handling recursive tree-shaped (and other kinds of) data structures—such as verifying that programs implementing advanced search data structures like AVL-trees or red-black trees indeed assure their defining properties, including balancedness—is currently beyond the capabalities of the common program verifiers associated with specification languages like JML [6] or Spec# [3]. These systems can verify in a semi-automatic way (as the user has to provide loop invariants) simpler properties like absence of null-pointer exceptions only. There are approaches, such as [15] or [13], considering verification of even the complex properties of the advanced data structures via testing or model checking, but these approaches are unsound as they work with bounded sets of instances of the data structures only.

Research on possibilities of sound verification of programs that handle complex tree-like structures has attracted researchers with various backgrounds, such as static analysis [19,23], proof theory [7], and formal language theory [17,5]. The approach that is the closest to ours is probably the one of PALE (Pointer Assertion Logic Engine) [17], which consists in translating the verification problem into the logic SkS [22] and using tree automata (although the classical ones only) to solve it. Our approach resembles PALE also in that we expect the user to provide the pre-, post-conditions, and the loop invariants, and that we reduce the validity problem for Hoare triples to the language emptiness problem. However, the use of the novel class of tree automata with arithmetic guards allows us to encode quantitative properties such as tree balancing that are not tackled in PALE.

In [23], a specialised framework of quantitative shape analysis based on abstract interpretation is introduced in order to verify *manipulation of AVL trees*. In [2], verification of some properties of *inserting into red-black trees* (including balancedness) is also reported. The work uses graph rewriting systems for describing the insertion procedure—the model is manually constructed. Then, an overapproximation using Petri graphs (Petri Nets with additional hypergraph structure) is used for verifying the fact that two red nodes never appear in succession. Further, graph type systems are used to check the balancedness. Not all desirable safety properties are covered this way, and both of the steps require a significant user involvement.

Recently, [16] has proposed an approach for verifying algorithms on balanced trees (and, in particular, on red-black trees) based on decidable theories of term algebras with Presburger arithmetic. These theories allow one to define functions from terms to integers, e.g., the maximal number of black nodes in paths from the root to a leave in a tree. The framework of [16], however, does not allow one to express local updates at an arbitrary control location, which consequently leads to a necessity of using an informal induction when proving program verification conditions. In other recent work [18], a different formal model – in particular, an extension of Separation Logic [**?**] with user-definable recursive shape predicates – is used to reason about safety of pointer-manipulating programs including insertion for red-black trees. This approach also targets the verification of Hoare triples in presence of user-specified program invariants. However, checking the verification conditions in this setting is done via sound, but incomplete proof rules, since the decidability status of the underlying logic is unknown.

The definition of TASC is a result of searching for a class of counter tree automata that combines interesting closure properties (union, intersection, complementation) with decidability of the emptiness problem. Existing works on extending tree automata with counters (e.g., [12,25]) have mostly concentrated on *in-breadth* counting of nodes with applications on verifying consistency of XML documents. Our work gives the possibility of *in-depth* counting in order to express balancing of recursive tree structures. It is worth noticing that similar computation models, such as alternating multi-tape and counter automata, have undecidable emptiness problems in the presence of two or more 1-letter input tapes, or, equivalently, non-increasing counters [20][1]. However, restricting the number of counters is problematic for obtaining the closure of automata under intersection. The solution we adopt here is to let the actions of the counters depend exclusively on the input tree alphabet, in other words, to encode them directly in the input as size functions. This solution can be seen as a generalisation of the visibly pushdown languages [1] to trees, for singleton stack alphabets. A similar approach has been recently taken in [10], where visibly tree automata with memory (VTAM) have been introduced. VTAM define a subclass of tree automata with one memory [8] enjoying boolean closure properties. Red-black trees and other balanced tree sets can be recognised using this formalism. However, the work in [10] does not directly address the verification of tree-manipulating programs, as it does not give a method to represent the effect of program statements on a set of trees represented as a tree automaton.

Roadmap. In Section 2, we summarise our verification methodology and describe our case study of insertion into red-black trees, which we will use in the paper. In Section 3, we introduce the notion of tree automata with size constraints. Section 4 provides results on determinisation of TASC, discusses their closure properties, and shows that their emptiness is decidable. In Section 5, it is shown how the semantics of tree manipulating programs can be encoded using TASC. Section 6 describes the use of TASC within the chosen case study. Finally, Section 7 contains some concluding remarks, including the future work.

[1] This result improves on the early work on alternating multi-tape automata recognising 1-letter languages in [14].

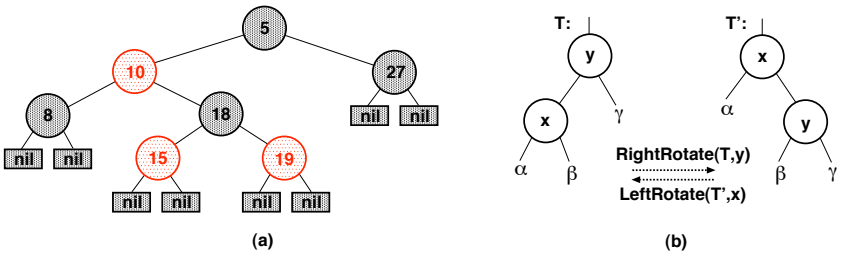

Fig. 1 (a) A red-black tree—nodes 10, 15, 19 are red, (b) the left and right tree rotation

2 A TASC-based Verification Methodology and a Running Example

In this section, we introduce our verification methodology for programs using balanced trees. In practice, several data structures based on balanced trees are commonly used, e.g., AVL trees. Here, we will use *red-black trees* as our running example. Red-black trees are binary search trees whose nodes are coloured by red or black. They are approximately balanced by constraining the way nodes can be coloured. The constraints insure that no maximal path can be more than twice longer than any other path.

More precisely, red-black trees are binary search trees whose nodes contain an element of an ordered data domain, a colour, a left and right pointer, and a pointer to its parent, and that satisfy the following properties:

1. Every node is either red or black.
2. The root is black.
3. Every leaf is black.
4. If a node is red, both its children are black.
5. Each path from the root to a leaf contains the same number of black nodes.

An example of a red-black tree is given in Figure 1 (a). The main operations on balanced trees (and hence also red-black trees) are searching, insertion, and deletion. When implementing the last two operations, one has to make sure that the trees remain balanced. This is usually done using tree rotations—cf. Figure 1 (b), which, in the case of red-black trees, can change the number of black nodes on a given path.

Because of the last condition on red-black trees mentioned above (i.e., having the same number of black nodes in each path), it is obvious that the set of red-black trees is not regular, i.e., not recognisable by standard tree automata [9]. Therefore, we have to introduce a tree automata model able to describe sets of (heap) configurations containing balanced trees. This model has to be powerful enough to describe these trees while still having properties allowing for automatic verification (i.e., decidability of inclusion, closure under some operations, etc.).

Here, we define such a class of extended tree automata—namely, tree automata with size constraints (TASC). We suppose the data content of the nodes to be abstracted away—we do not verify sortedness. Basic program blocks (i.e., individual program statements or groups of statements that we view as atomic like, e.g., rotations) define effective transformations on TASC.

We assume the user to specify the precondition and postcondition of the program to be verified. Further, we suppose the user to supply an invariant for each loop. The preconditions and postconditions as well as loop invariants are specified by TASC. Then, the verification is performed by automatically checking the validity of each triple $\{P\}\ \mathbf{C}\ \{Q\}$, where:

- P is the program precondition or a loop invariant,
- Q is the program postcondition or a loop invariant, and
- C is a loop-free fragment of the code between P and Q.

This is done by computing the image of the precondition after an application of the code of the program block and by checking that the image implies the postcondition. This check is done using language inclusion for TASC.

In Figure 2, we give the pseudo-code of the inserting operation for red-black trees [11]. For this program, we want to show that after an insertion of a node, a red-black tree remains a red-black tree. In our work, we restrict ourselves to calculating the effects of program blocks which preserve the tree structure of the heap. This is not the case in general since pointer operations can temporarily break the tree structure, e.g., in the code for performing a rotation. The operations that we handle are the following:

1. tests on the tree structure (like `x->parent == x->parent->parent->left`),
2. changing data of a node (as, e.g., recolouring of a node `x->colour = red`),
3. left and right rotations (Figure 1 (b)),
4. moving a pointer up or down a tree structure (like `x = x->parent->parent`),
5. low-level insertion/deletion, i.e., the physical addition/removal of a terminal node, that is then followed by re-balancing operations.

3 Tree Automata with Size Constraints

In what follows, we work with the set $\mathfrak{D}$ of all boolean combinations of formulae of the form $x-y\diamond c$ or $x\diamond c$, for some $c\in\mathbb{Z}$ and $\diamond\in\{\leq,\geq\}$. We introduce equality as $x-y=c\ :\ x-y\leq c\wedge x-y\geq c$. Notice that negation can be eliminated from any formula of $\mathfrak{D}$ since $x-y\not\leq c\iff x-y\geq c+1$. Also, any constraint of the form $x-y\geq c$ can be equivalently written as $y-x\leq -c$. For a closed formula φ, we write $\models\varphi$ to denote that φ is valid, i.e., equivalent to true.

A *ranked alphabet* Σ is a set of symbols together with a function $\#:\Sigma\to\mathbb{N}$. For $f\in\Sigma$, the value $\#(f)$ is said to be the *arity* of f. Symbols of zero arity are referred to as *constants*. We denote by Σ_n the set of all symbols of arity n from Σ. Let λ denote the empty sequence. A *tree* t over an alphabet Σ is a partial mapping $t:\mathbb{N}^*\to\Sigma$ that satisfies the following conditions:

- $dom(t)$ is a finite prefix-closed subset of $\mathbb{N}^*$, and
- for each $p\in dom(t)$, if $\#(t(p))=n>0$, then $\{i\mid pi\in dom(t)\}=\{1,\ldots,n\}$.

A special case of a ranked alphabet is the *binary alphabet* in which all symbols have arities either zero or two. Trees over binary alphabets are referred to as *binary trees*.

A *subtree* of t starting at a position $p\in dom(t)$ is a tree $t_{|p}$ defined as $t_{|p}(q)=t(pq)$ if $pq\in dom(t)$, and undefined otherwise. Given a set of positions $P\subseteq\mathbb{N}^*$,

```
RB-Insert(T,x):
  Tree-Insert(T,x);   % Inserts a new leaf node x
  x->colour = red;
  while (x != root && x->parent->colour == red) {
    if (x->parent == x->parent->parent->left) {
      if (x->parent->parent->right->colour == red) {
        x->parent->colour = black;                          % Case 1
        x->parent->parent->right->colour = black;
        x->parent->parent->colour = red;
        x = x->parent->parent;
      }
      else {
        if (x == x->parent->right) {                        % Case 2
          x = x->parent;
          LeftRotate(T,x);
        }
        x->parent->colour = black;                          % Case 3
        x->parent->parent->colour = red;
        RightRotate(T,x->parent->parent);
      }
    }
    else .... % the same as above with right and left exchanged
  }
  root->colour = black;
```

Fig. 2 A procedure for inserting into red-black trees

we define the *frontier* of P as the set $fr(P) = \{p \in P \mid \forall i \in \mathbb{N}\ pi \notin P\}$ i.e., the set of tree positions from P whose direct successors are not in P any longer. For a tree t, we use $fr(t)$ as a shortcut for $fr(dom(t))$. If t is a tree and $\mathbf{p} = \langle p_1, \ldots, p_n \rangle$ is a sequence of positions $p_i \in dom(t)$, we denote by $t \bullet_{\mathbf{p}} \langle t_1, \ldots, t_n \rangle$ the result of replacing each subtree $t_{|p_i}$ by t_i for all $1 \leq i \leq n$. We denote by $T(\Sigma)$ the set of all trees over the alphabet Σ.

Intuitivelly, a *tree mapping* is a generalisation of a homomorphism that maps each position from the domain of the source tree into a subtree of the destination tree:

Definition 1 Given two trees $t : \mathbb{N}^* \to \Sigma$ and $t' : \mathbb{N}^* \to \Sigma'$, a function $h : dom(t) \to dom(t')$ is said to be a *tree mapping between t and t'* if the following holds:

- $h(\lambda) = \lambda$, and
- for any $p \in dom(t)$, if $\#(t(p)) = n > 0$, then there exists a prefix-closed set $Q \subseteq \mathbb{N}^*$ such that $pQ \subseteq dom(t')$ and $h(pi) \in fr(pQ)$ for all $1 \leq i \leq n$.

A *size function* (or measure) associates to every tree $t \in T(\Sigma)$ an integer $|t| \in \mathbb{Z}$. Size functions are defined inductively on the structure of the tree. For each $f \in \Sigma$, if $\#(f) = 0$, then $|f|$ is a constant c_f, otherwise, for $\#(f) = n$, we have:

$$|f(t_1, \ldots, t_n)| = \begin{cases} b_1|t_1| + c_1 \text{ if } \models \delta_1(|t_1|, \ldots, |t_n|) \\ \quad \ldots \\ b_n|t_n| + c_n \text{ if } \models \delta_n(|t_1|, \ldots, |t_n|) \end{cases}$$

where $b_1, \ldots, b_n \in \{0, 1\}$, $c_1, \ldots, c_n \in \mathbb{Z}$, and $\delta_1, \ldots, \delta_n \in \mathfrak{D}$, all depending on f. In order to have a consistent definition, it is required that $\delta_1, \ldots, \delta_n$ define a partition of $\mathbb{N}^n$, i.e., $\models \forall x_1 \ldots \forall x_n \bigvee_{1 \leq i \leq n} \delta_i(x_1, \ldots, x_n) \wedge \bigwedge_{1 \leq i < j \leq n} \neg(\delta_i(x_1, \ldots, x_n) \wedge$

$\delta_j(x_1,\ldots,x_n)$).[2] A *sized alphabet* $(\Sigma,|.|)$ is a ranked alphabet with an associated size function.

Example. The height of a binary tree is an example of a tree measure, defined as $|c| = 1$, if $\#(c) = 0$, and

$$|f(t_1,t_2)| = \begin{cases} |t_1|+1 \text{ if } |t_1| \geq |t_2| \\ |t_2|+1 \text{ if } |t_2| < |t_1| \end{cases}$$

if $\#(f) = 2$.

A *tree automaton with size constraints* (TASC) over a sized alphabet $(\Sigma,|.|)$ is a 3-tuple $A = (Q,\Delta,F)$ where Q is a finite set of states, $F \subseteq Q$ is a designated set of final states, and Δ is a set of transition rules of the form $f(q_1,\ldots,q_n) \xrightarrow{\varphi(|1|,\ldots,|n|)} q$, where $f \in \Sigma$, $\#(f) = n$, and $\varphi \in \mathfrak{D}$ is a formula with n free variables. For constant symbols $a \in \Sigma$, $\#(a) = 0$, the automaton has unconstrained rules of the form $a \rightarrow q$.

A *run* of A over a tree $t : \mathbb{N}^* \rightarrow \Sigma$ is a mapping $\pi : dom(t) \rightarrow Q$ such that, for each position $p \in dom(t)$, where $q = \pi(p)$, we have:

- if $\#(t(p)) = n > 0$ and $q_i = \pi(pi)$, $1 \leq i \leq n$, then Δ has a rule $t(p)(q_1,\ldots,q_n) \xrightarrow{\varphi(|1|,\ldots,|n|)} q$ and $\models \varphi(|t_{|p1}|,\ldots,|t_{|pn}|)$,
- otherwise, if $\#(t(p)) = 0$, then Δ has a rule $t(p) \rightarrow q$.

A run π is said to be *accepting* if and only if $\pi(\lambda) \in F$. As usual, the *language* of A, denoted as $\mathscr{L}(A)$ is the set of all trees over which A has an accepting run.

Example. The following TASC recognises the set of all balanced red-black trees. Let $\Sigma = \{red, black, null\}$ with $\#(red) = \#(black) = 2$ and $\#(null) = 0$. First, we define the size function to be the maximal number of black nodes from the root to a leaf: $|null| = 1$, $|red(t_1,t_2)| = max(|t_1|,|t_2|)$, and $|black(t_1,t_2)| = max(|t_1|,|t_2|)+1$. The TASC recognising the set of all balanced red-black trees may now be defined as $A_{rb} = (\{q_b,q_r\},\Delta,\{q_b\})$ with the set of transition rules:

$$\Delta = \{null \rightarrow q_b, black(q_{b/r},q_{b/r}) \xrightarrow{|1|=|2|} q_b, red(q_b,q_b) \xrightarrow{|1|=|2|} q_r\}$$

By using $q_{x/y}$ within the left-hand side of a transition rule, we mean the set of rules in which either q_x or q_y take the place of $q_{x/y}$. □

Finally, for *binary* trees only, we define the notion of *balance*. For a given binary tree t and a position $p \in dom(t)$, we define the balance of t at p as the difference $|t_{|p0}| - |t_{|p1}|$ between the sizes of the left and right subtrees of p.

[2] For technical reasons related to the decidability of the emptiness problem for TASC, we do not allow arbitrary linear combinations of $|t_i|$ in the definition of $|f(t_1,\ldots,t_n)|$.

4 Closure Properties and Decidability of TASC

This section is devoted to the closure of the class of TASC under the operations of union, intersection, and complement. The decidability of the emptiness problem is also proved.

4.1 Determinisation

A TASC is said to be *deterministic* if, for every input tree, the automaton has at most one run. For every TASC A, we can effectively construct a deterministic TASC A_d such that $\mathscr{L}(A) = \mathscr{L}(A_d)$. We adapt the classical subset construction for determinising bottom-up tree automata. We have to take into account the fact that in a deterministic TASC, two rules which have the same left-hand side should not be applicable simultaneously. This problem is solved below by constructing guards of transition rules of the deterministic TASC as conjunctions of the original transition guards, which could otherwise be in a conflict, and their negations in all possible combinations. This way, we ensure that all transitions with the same left-hand side have guards that can never be satisfied simultaneously.

Concretely, let $A = (Q, \Delta, F)$. We define $A_d = (Q_d, \Delta_d, F_d)$ where $Q_d = \mathscr{P}(Q)$, $F_d = \{s \in Q_d \mid s \cap F \neq \emptyset\}$, and $f(s_1, \ldots, s_n) \xrightarrow{\varphi} s \in \Delta_d$ if and only if:

$$s \subseteq \{q | f(q_1, \ldots, q_n) \xrightarrow{\psi} q \in \Delta, q_i \in s_i\}, \text{ and } s \neq \emptyset$$

$$\varphi = \bigwedge \{\psi | f(q_1, \ldots, q_n) \xrightarrow{\psi} q \in \Delta, q_i \in s_i,\ q \in s\} \wedge \bigwedge \{\neg\psi | f(q_1, \ldots, q_n) \xrightarrow{\psi} q \in \Delta, q_i \in s_i,\ q \notin s\}$$

In the case of transition rules involving constant symbols, we have $a \rightarrow s \in \Delta_d$ if and only if $s = \{q \mid a \rightarrow q \in \Delta\}$. The following theorem proves that non-deterministic and deterministic TASC recognise exactly the same languages.

Theorem 1 *A_d is deterministic and $\mathscr{L}(A_d) = \mathscr{L}(A)$.*

Proof (1) To prove that A_d is deterministic, suppose $t \xrightarrow[A_d]{*} s$ and $t \xrightarrow[A_d]{*} s'$, for some $t \in T(\Sigma)$ and two states $s, s' \in Q_d$. We prove $s = s'$ by induction on the structure of t. If $t = a \in \Sigma_0$, we have $s = s' = \{q \in Q \mid a \xrightarrow[A]{} q\}$ by definition of A_d. Otherwise, let $t = f(t_1, \ldots, t_n)$ for some $f \in \Sigma_n$ and $t_1, \ldots, t_n \in T(\Sigma)$, and, by induction hypothesis, there exist unique states $s_i \in Q_d$ such that $t_i \xrightarrow[A_d]{*} s_i$, $1 \leq i \leq n$. Suppose that $s \neq s'$, that is, there exists a state $q \in Q$ which either belongs to s and does not belong to s' or vice-versa. Let us consider the first case, the other one being symmetric. By the definition of A_d, Δ_d has two rules $f(s_1, \ldots, s_n) \xrightarrow{\varphi} s$ and

$f(s_1,\ldots,s_n) \xrightarrow{\varphi'} s'$, and A has a rule $f(q_1,\ldots,q_n) \xrightarrow{\psi} q$, for some $q_i \in s_i$, $1 \le i \le n$, such that $\varphi \Rightarrow \psi$ and $\varphi' \Rightarrow \neg\psi$. But since s and s' are reachable from t in A_d, it must be the case that $\models \varphi(|t_1|,\ldots,|t_n|)$ and $\models \varphi'(|t_1|,\ldots,|t_n|)$, which leads to a contradiction. Hence $s = s'$.

(2) "$\mathscr{L}(A_d) \subseteq \mathscr{L}(A)$". We prove inductively that, for all $t \in T(\Sigma)$ and $s \in Q_d$ such that $t \xrightarrow[A_d]{*} s$, for all $q \in s$, we have $t \xrightarrow[A]{*} q$. If $t = a \in \Sigma_0$, by definition of A_d, we have $s = \{q \mid a \xrightarrow[A]{} q\}$. Otherwise, $t = f(t_1,\ldots,t_n)$ for some $f \in \Sigma_n$ and $t_1,\ldots,t_n \in T(\Sigma)$, and $t_i \xrightarrow[A_d]{*} s_i$, $1 \le i \le n$. By induction hypothesis, for all $q_i \in s_i$, we have $t_i \xrightarrow[A]{*} q_i$. By definition of A_d, there exists a rule $r : f(s_1,\ldots,s_n) \xrightarrow{\varphi} s$ such that, for each rule $f(q_1,\ldots,q_n) \xrightarrow{\psi} q$ with $q_i \in s_i$ and $q \in s$, we have $\varphi \Rightarrow \psi$. Moreover, the rule r is applicable for the subtrees $t_1,\ldots,t_n$, i.e., $\models \varphi(|t_1|,\ldots,|t_n|)$. Hence, each rule $f(q_1,\ldots,q_n) \xrightarrow{\psi} q$ is applicable. Therefore, for all $q \in s$, we have $t \xrightarrow[A]{*} q$. If $s \in F_d$, then, by the definition of A_d, there exists $q \in s \cap F$. Thus t is accepted by A if it is accepted by A_d.

"$\mathscr{L}(A_d) \supseteq \mathscr{L}(A)$". We prove inductively that, for all $t \in T(\Sigma)$ and $q \in Q$, if $t \xrightarrow[A]{*} q$, then there exists $s \in Q_d$ such that $t \xrightarrow[A_d]{*} s$ and $q \in s$. If $t = a \in \Sigma_0$, we have $s = \{q \mid a \to q\}$ and $\varphi = \top$. Otherwise, $t = f(t_1,\ldots,t_n)$ for some $f \in \Sigma_n$ and $t_1,\ldots,t_n \in T(\Sigma)$, and $t_i \xrightarrow[A]{*} q_i$, for some $q_i \in Q$, $1 \le i \le n$. By the induction hypothesis, there exist some $s_i \in Q_d$ such that $t_i \xrightarrow[A_d]{*} s_i$ and $q_i \in s_i$. Also, if $t \xrightarrow[A]{*} f(q_1,\ldots,q_n) \xrightarrow[A]{\psi} q$, then $\models \psi(|t_1|,\ldots,|t_n|)$. Consider now the set of guards $\mathscr{G} = \{\psi' \mid \exists q_1 \in s_1,\ldots,\exists q_n \in s_n \exists q' \in Q.\ f(q_1,\ldots,q_n) \xrightarrow[A]{\psi'} q'\}$, and Γ_ψ be the set of all subsets of $\mathscr{G}$ that contain ψ. For any set of guards $\mathscr{I}$, we denote by $\Psi_\mathscr{I}$ the formula $\bigwedge_{\varphi\in\mathscr{I}} \varphi \wedge \bigwedge_{\varphi\in\mathscr{G}\setminus\mathscr{I}} \neg\varphi$. Obviously, $\psi = \bigvee_{\mathscr{I}\in\Gamma_\psi} \Psi_\mathscr{I}$. Since $\models \psi(|t_1|,\ldots,|t_n|)$, there exists some $\mathscr{I} \in \Gamma_\psi$ such that $\models \Psi_\mathscr{I}(|t_1|,\ldots,|t_n|)$. Now, let $s = \{q' \mid \exists q_1 \in s_1,\ldots, \exists q_n \in s_n.\ f(q_1,\ldots,q_n) \xrightarrow[A]{\psi'} q',\ \psi' \in \mathscr{I}\}$, and $\varphi = \Psi_\mathscr{I}$. Notice that $q \in s$. By the definition of A_d, there exists a rule $f(s_1,\ldots,s_n) \xrightarrow{\varphi} s$ in Δ_d, and, moreover, it is applicable, hence $t \xrightarrow[A_d]{*} s$. By the definition of A_d, if $q \in F$, then $s \in F_d$, hence t is accepted by A_d if it is accepted by A. □

Let us have two arbitrary TASCs $A_1 = (Q_1, \Delta_1, F_1)$ and $A_2 = (Q_2, \Delta_2, F_2)$. We can assume w.l.o.g. that Q_1 and Q_2 are disjoint. Let $A_1 \cup A_2 = (Q_1 \cup Q_2, \Delta_1 \cup \Delta_2, F_1 \cup F_2)$.

Lemma 1 *Given a sized alphabet Σ and two TASCs $A_i = (Q_i, \Delta_i, F_i)$, $i = 1,2$, over Σ, we have $\mathscr{L}(A_1 \cup A_2) = \mathscr{L}(A_1) \cup \mathscr{L}(A_2)$.*

Proof As in the standard case of tree automata, if $t \in \mathscr{L}(A_1 \cup A_2)$, then $A_1 \cup A_2$ has an accepting run $\pi : dom(t) \to Q_1 \cup Q_2$ over t. Since $Q_1 \cap Q_2 = \emptyset$, we can prove by induction on the structure of t that either (1) $\pi(dom(t)) \subseteq Q_1$ or (2) $\pi(dom(t)) \subseteq Q_2$. In the first case, we have $t \in \mathscr{L}(A_1)$, whereas in the second, we have $t \in \mathscr{L}(A_2)$, therefore $\mathscr{L}(A_1 \cup A_2) \subseteq \mathscr{L}(A_1) \cup \mathscr{L}(A_2)$. The other direction is trivial. □

A TASC $A = (Q, \Delta, F)$ is said to be *complete* if, for any tree $t \in T(\Sigma)$, there exists a state $q \in Q$ such that $t \xrightarrow[A]{*} q$. An arbitrary TASC can be completed by adding a sink state $\sigma \notin Q$ and the following rules, for all $f \in \Sigma$, $q_1, \ldots, q_n \in Q$, where $n = \#(f)$:

$$f(q_1, \ldots, q_n) \xrightarrow{\varphi} \sigma \in \Delta_c \text{ iff } \varphi = \bigwedge\{\neg\psi \mid f(q_1, \ldots, q_n) \xrightarrow{\psi} q \in \Delta\}$$
$$f(q_1, \ldots, \sigma, \ldots q_n) \xrightarrow{\top} \sigma \in \Delta_c$$

Above, Δ_c denotes the set Δ to which the new transition rules have been added. The complete TASC is $A_c = (Q \cup \{\sigma\}, \Delta_c, F)$. Notice that if there are no rules $f(q_1, \ldots, q_n) \xrightarrow[A]{\psi} q$, then there is a rule $f(q_1, \ldots, q_n) \xrightarrow[A_c]{\top} q$. Note that if A is deterministic, so is A_c.

Lemma 2 *Given a sized alphabet Σ and a TASC $A = (Q, \Delta, F)$ over Σ, we have $\mathscr{L}(A_c) = \mathscr{L}(A)$.*

Proof Since the set of transition rules of A_c is a superset of Δ, we have $\mathscr{L}(A_c) \supseteq \mathscr{L}(A)$. By contradiction, suppose that there exists a tree $t \in \mathscr{L}(A_c) \setminus \mathscr{L}(A)$. Then A_c has an accepting run π_c on t, which uses at least one of the newly added rules. But, since all the rules of A_c which are not in Δ lead to σ, and all rules where σ occurs on the left-hand side must have it on the right-hand side also, then $\pi_c(\lambda) = \sigma$. However, σ is not an accepting state of A_c, which contradicts the assumption that π_c is an accepting run of A_c. □

The *complement* of a deterministic complete TASC $A = (Q, \Delta, F)$ is defined as $\overline{A} = (Q, \Delta, Q \setminus F)$.

Lemma 3 *Given a sized alphabet Σ and a complete deterministic TASC $A = (Q, \Delta, F)$ over Σ, we have $t \in \mathscr{L}(A)$ if and only if $t \notin \mathscr{L}(\overline{A})$ for any $t \in T(\Sigma)$.*

Proof If A is complete and deterministic, then for each $t \in T(\Sigma)$, A has exactly one run $\pi : dom(t) \to Q$. If $t \in \mathscr{L}(A)$, then π is accepting, and $\pi(\lambda) \in F$. In this case, π is not accepting for $\overline{A}$, hence $t \notin \mathscr{L}(\overline{A})$. The other direction is symmetric. □

Since we can construct automata for union and complement of TASC, it is possible to define intersection as $A_1 \cap A_2 = \overline{\overline{A_1} \cup \overline{A_2}}$.

4.3 Deciding Emptiness

This section is dedicated to the decidability proof for TASC. We show that all runs of a TASC are in direct correspondence to the accepting runs of an effectivelly constructed Alternating Pushdown System (APDS). The existence of accepting runs for APDS is a well-known decidable problem, which occurs as a consequence of the results in [4]. Namely, it is shown that, given a regular set C of configurations (pairs of the form $\langle q, w \rangle$, where q is a control state, and w is the contents of the stack), the set $pre^*(C)$ of all predecessor configurations is also regular and can be effectivelly computed from C. In particular, the set $pre^*_q(C) = \{w \mid \langle q, w \rangle \in pre^*(C)\}$ is also regular, and effectively computable from C. In other words, if the APDS has a run leading from a control state q_0 into a state in C if and only if the set $pre^*_{q_0}$ is not empty. Since the latter is a regular set (recognized by an alternating automaton) its emptiness is decidable. This entails the decidability of the emptiness problem for APDS.

Given an arbitrary TASC, we translate it into an APDS whose stack encodes the value of one integer counter, denoted by y from now on. An APDS is a 4-tuple $S = (Q, \Gamma, \delta, F)$ where:

- Q is a finite set of control locations,
- Γ is a finite stack alphabet,
- $F \subseteq Q$ is a set of final control locations,
- δ is a mapping from $Q \times \Gamma$ into $\mathscr{P}(\mathscr{P}(Q \times \Gamma^*))$.

Notice that an APDS does not have an input alphabet since we are interested in the behaviours it generates, rather than in the accepted language. A run of an APDS is a tree $t : \mathbb{N}^* \to (Q \times \Gamma^*)$ satisfying the following property: for any $p \in dom(t)$, if $t(p) = \langle q, \gamma w \rangle$, then $\{t(pi) \mid 1 \leq i \leq \#(t(p))\} = \{\langle q_1, w_1 w \rangle, \ldots, \langle q_n, w_n w \rangle\}$, where $\{\langle q_1, w_1 \rangle, \ldots, \langle q_n, w_n \rangle\} \in \delta(q, \gamma)$. The run is accepting if all control locations occurring on its frontier are final.

For a TASC $A = (Q, \Delta, F)$ over a sized alphabet $(\Sigma, |.|)$, let $S_A = (Q_A, \Gamma, \delta_A, F_A)$ be the APDS where $Q_A = Q \times \Sigma \cup \Pi$, $\Gamma = \{-, 0, 1\}$, and $F_A = \{q_f\} \subset \Pi$. Here, Π is an additional set of states that are needed in the construction of S_A from A and that are not of the form $\langle q, f \rangle$. We use 0 as the beginning of the stack marker, $-$ on top of the stack denotes a negative value, and 1 is used for the unary encoding of the absolute value of the counter. We represent an integer value $n \in \mathbb{Z}$ using the unary encoding:

$$(n)_1 = \begin{cases} 1^n 0, & \text{if } n \geq 0 \\ -1^{-n} 0 & \text{if } n < 0 \end{cases}$$

The primitive operations on the counter y, i.e., increment, decrement, and zero test, are encoded by the moves given in Figure 3. For example, if the value of y in a control state q is -2, a transition that increments y and moves into q' is simulated by the following sequence of moves: $\langle q,-110\rangle \rightsquigarrow \langle q^-,110\rangle \rightsquigarrow \langle q'^-,10\rangle \rightsquigarrow \langle q',-10\rangle$. Note that $(-2)_1 = -110$ and $(-1)_1 = -10$.

$q \xrightarrow{y'=y+1} q'$	$q \xrightarrow{y'=y-1} q'$	$q \xrightarrow{y=0} q'$
$\langle q,1\rangle \hookrightarrow \langle q',11\rangle$		
$\langle q,0\rangle \hookrightarrow \langle q',10\rangle$		
$\langle q,-\rangle \hookrightarrow \langle q^-,\varepsilon\rangle$	$\langle q,1\rangle \hookrightarrow \langle q',\varepsilon\rangle$	
$\langle q^-,1\rangle \hookrightarrow \langle q'^-,\varepsilon\rangle$	$\langle q,0\rangle \hookrightarrow \langle q',-10\rangle$	$\langle q,0\rangle \hookrightarrow \langle q',0\rangle$
$\langle q'^-,1\rangle \hookrightarrow \langle q',-1\rangle$	$\langle q,-\rangle \hookrightarrow \langle q',-1\rangle$	
$\langle q'^-,0\rangle \hookrightarrow \langle q',0\rangle$		

Fig. 3 Encoding a counter by a stack

Let $Perm(N)$ denote the set of all permutations $I : \{1,\ldots,N\} \rightarrow\{ 1,\ldots,N\}$. For technical reasons, the following lemma is needed in the rest of the section.

Lemma 4 *Every formula $\varphi(x_1,\ldots,x_N)$ of $\mathfrak{D}$ can be effectively written as a disjunction of formulae of the following form, for a suitable permutation $I \in Perm(N)$ of its free variables :*

$$\bigwedge_{k=1}^{N-1} x_{I(k)} - x_{I(k+1)} \diamond_k c_k \ \wedge \bigwedge_{m\in M\subseteq\{1,\ldots,N\}} x_m \leq d_m \ \wedge \bigwedge_{p\in P\subseteq\{1,\ldots,N\}} x_p \geq e_p$$

where $\diamond_k \in \{\leq,=\}$ and $c_k, d_m, e_p \in \mathbb{Z}$.

Proof First, we eliminate all occurences of negation and $\geq$. Second, we replace any conjunction of the form $c_1 \leq x-y \leq c_2$ for $c_1 < c_2$ (for $c_1 > c_2$, the conjunction is not satisfiable and the original formula can be simplified accordingly), by the disjunction $\bigvee_{c\in\{c_1,c_1+1,\ldots,c_2\}} x = y + c$. Third, we put the resulting formula in DNF and process each disjunct as follows.

For each permutation $I \in Perm(N)$ of the free variables in φ, we define the induced ordering $\theta_I \ : \ x_{I(1)} \leq x_{I(2)} \leq \ldots \leq x_{I(N)}$. Let $\Theta = \bigvee_{I\in Perm(N)} \theta_I$ be the (logically valid) disjunction of all possible orderings of the free variables $x_1,\ldots,x_N$. In the following, we work with the DNF form of $\varphi \wedge \Theta$, in which each disjunct is necessarily associated with some ordering. We transform each clause (disjunct) $\theta_I \wedge \psi$ of the DNF form of $\varphi \wedge \Theta$ by applying one of the four cases below for each constraint $x_i - x_j \diamond c$, $\diamond \in \{\leq,=\}$, occurring in ψ:

1. If $\theta_I \Rightarrow x_i \leq x_j$ and $c \leq 0$, then there exist $x_i = x_{I(k)} \leq x_{I(k+1)} \leq \ldots \leq x_{I(l)} = x_j$ in θ_I. Let $C = \{\langle c_k,\ldots,c_{l-1}\rangle \mid c_i \geq 0, k \leq i < l, \ \sum_{i=k}^{l-1} c_i = c\}$. Since C is finite, we can replace $x_i - x_j \diamond c$ by the equivalent formula $\bigvee_{\mathbf{c}\in C} \bigwedge_{k\leq i<l} x_{I(i)} - x_{I(i+1)} \diamond c_i$.
2. The case of $\theta_I \Rightarrow x_i \geq x_j$ and $c \geq 0$ is treated in a symmetric way with the first point.

3. If $\theta_I \Rightarrow x_i \leq x_j$ and $c > 0$, the constraint is trivially valid and can be eliminated from the clause. In the case where $x_i - x_j \diamond c$ is the only constraint in the clause, the original formula φ is valid.
4. If $\theta_I \Rightarrow x_i \geq x_j$ and $c < 0$, we discard the entire clause $\theta_I \wedge \psi$ as unsatisfiable. In the case where this was the only clause, the original formula φ is unsatisfiable.

In the resulting formula, we replace:

- any conjunction of constraints of the form $x - y \leq c' \wedge x - y \leq c''$ by $x - y \leq \min(c', c'')$,
- any conjunction of constraints of the form $x - y = c' \wedge x - y = c'$ by simply $x - y = c'$,
- any conjunction of constraints of the form $x - y \leq c' \wedge x - y = c''$ by $x - y = c''$ if $c'' \leq c'$, and
- any conjunction containing a subformula of the form $x - y \diamond c' \wedge x - y = c''$ by $\bot$ if $c' < c''$.

□

We shall encode a move of A as a series of moves of S_A. As A moves bottom-up on the tree, S_A will perform a series of alternating top-down transitions, simulating the move of A in reverse. The stack (counter) of S_A is intended to encode the value of the size function $|.|$ at the current tree node.

Suppose that A has a transition rule $f(q_1, \ldots, q_n) \xrightarrow{\varphi} q$ and that the current node is of the form $f(t_1, \ldots, t_n)$ with $|f(t_1, \ldots, t_n)| = b_r|t_r| + c_r$, and δ_r is the disjunctive condition such that $\models \delta_r(|t_1|, \ldots, |t_n|)$, according to the definition of the size function (see Section 3). W.l.o.g., we consider from now on that φ and δ_r have the same set of free variables, denoted $x_1, \ldots, x_n$. In what follows, we consider the case $b_r = 1$, i.e., $|f(t_1, \ldots, t_n)| = |t_r| + c_r$. The case $b_r = 0$ can be treated in a similar way, by guessing the value $|t_r|$. The position r is said to be the *reference position* of the subtree $f(t_1, \ldots, t_n)$. The value $|t_r|$ is said to be the *reference value* of $f(t_1, \ldots, t_n)$.

Without losing generality, we consider that the difference constraint formula $\varphi \wedge \delta_r \in \mathfrak{D}$ has already been converted into the normal form of Lemma 4, that is, a disjunction of formulae of the form:

$$\bigwedge_{k=1}^{n-1} x_{I(k)} - x_{I(k+1)} \diamond_k d_k \ \wedge \bigwedge_{m \in M \subseteq \{1,\ldots,n\}} x_m \leq e_m \ \wedge \bigwedge_{p \in P \subseteq \{1,\ldots,n\}} x_p \geq l_p$$

where $\diamond_k \in \{\leq, =\}$, $d_k, e_m, l_p \in \mathbb{Z}$, and $I \in Perm(n)$. For the rest of this section, let us fix one such disjunct.

After each sequence of universal moves, S_A creates n copies of its counter y, let us name them $y_1, \ldots, y_n$. The counter y_i is intended to hold the value $|t_{I(i)}|$ for $1 \leq i \leq n$, and the counter y holds the value $|f(t_1, \ldots, t_n)|$. Let $i_r = I^{-1}(r)$ be the index of the counter y_{i_r} that holds the reference value of the given transition, i.e., $y = y_{i_r} + c_r$. With this notation, Figure 4 (a) shows the alternating moves of S_A that simulate the A-transition considered, for one disjunct of $\varphi \wedge \delta_r$. Figure 4 (b) shows the moves for transitions of the form $a \rightarrow q$.

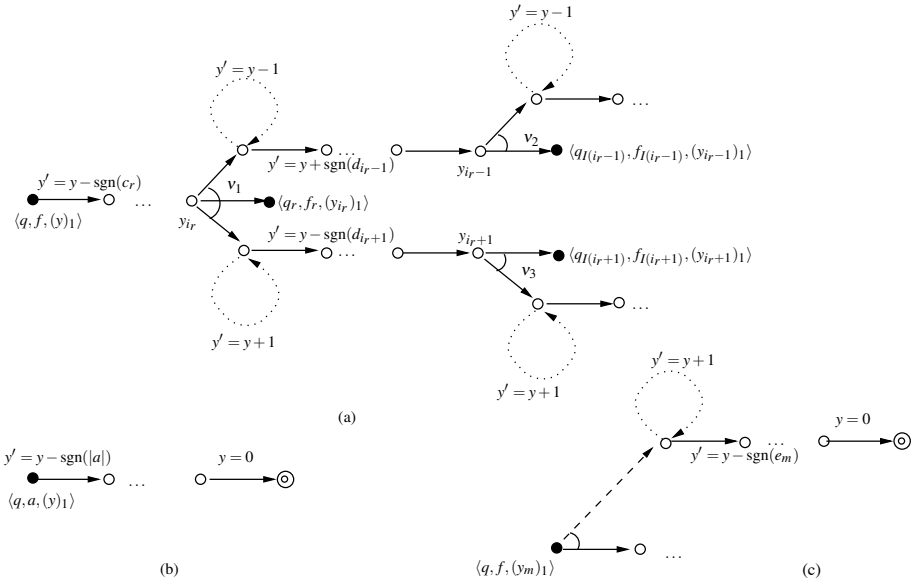

Fig. 4 Simulation of a TASC by an APDS

Filled circles in Figure 4 represent states from $Q \times \Sigma$, and empty circles are additional states from Π. The only accepting state of S_A, named q_f, is marked by a double circle. The notation $\mathrm{sgn}(\ldots)$ denotes the sign function, i.e., $\mathrm{sgn}(n) = 1$ if $n > 0$, $\mathrm{sgn}(0) = 0$, and $\mathrm{sgn}(n) = -1$ if $n < 0$. Next, $\nu_1, \nu_2, \ldots$ are symbolic names for the universal moves performed by S_A. Further, in what follows, we will denote a configuration $\langle\langle q, f\rangle, u\rangle$ of S_A by writing $\langle q, f, u\rangle$. In particular, in Figure 4, configurations from $Q \times \Sigma \times \Gamma^*$ are labeled by triples of the form $\langle q, f, (y)_1\rangle$. Here, $(y)_1$ denotes the unary encoding of the value of the y counter. Moreover, for simplicity, configurations from $\Pi \times \Gamma^*$ are labeled only with $(y)_1$ in Figure 4.

When simulating the A-transition $f(q_1, \ldots, q_n) \xrightarrow{\varphi} q$, S_A starts with the configuration $\langle q, f, (y)_1\rangle$ (cf. Figure 4 (a)). In order to derive the reference value y_{i_r} from y, S_A performs $|c_r|$ decrement or increment actions, depending on whether the sign of c_r is positive or negative. Then S_A performs the universal move ν_1 making three copies of itself (unless $i_r = 1$ when the upper branch is omitted and/or $i_r = n$ when the lower branch is omitted). The middle branch simply moves to the appropriate control state $\langle q_r, f_r\rangle$ with stack $(y_{i_r})_1$. The upper and lower branches are used to produce the values y_{i_r-1} and y_{i_r+1} if needed.

The upper branch of the universal move ν_1 depicted in Figure 4 depends on $\diamond_r \in \{\leq, =\}$. If $\diamond_r$ is $=$, then S_A performs a sequence of increment/decrement operations of length d_{i_r-1} in order to obtain the value y_{i_r-1} from y_{i_r} (since $y_{i_r-1} = y_{i_r} + d_{i_r-1}$). If $\diamond_r$ is $\leq$, then there is an additional existential (non-deterministic) transition—depicted using a dotted arrow in Figure 4 (a)—which decrements the counter an arbitrary number of times in order to obtain a smaller value (since $y_{i_r-1} \leq y_{i_r} + d_{i_r-1}$).

A similar sequence of transitions is performed by the lower branch of ν_1. Note that the symbols $f_{I(i_r-1)}, f_r, f_{I(i_r+1)}$ are chosen arbitrarily, that is, for each triple $(g_1,g_2,g_3) \in \Sigma_n^3$, S_A performs three universal moves that are identical to ν_1, ν_2, ν_3, with g_1, g_2, and g_3 substituted for $f_{I(i_r-1)}, f_r$, and $f_{I(i_r+1)}$, respectively.

Next, if $i_r - 1 > 1$, the simulation continues with the binary universal move ν_2. The lower branch of ν_2 changes the control into $\langle q_{I(i_r-1)}, f_{I(i_r-1)}\rangle$ without changing the stack. The upper branch of ν_2 leads to a control state from Π, from which the remaining values $y_{i_r-2},\ldots,y_1$ are produced. Symmetrically, the universal move ν_3 leads to configurations producing the values $y_{i_r+1},\ldots,y_n$.

Clearly, the values of the counters $y_1, y_2, \ldots, y_n$ that are obtained in the way described above will satisfy the constraint $\varphi \wedge \delta_r$ when used as the sizes of the subtrees $t_{I(1)}, t_{I(2)}, \ldots, t_r, \ldots, t_{I(n)}$. Moreover, at the same time, any assignment satisfying this formula can be obtained in some run of S_A by iterating the increment/decrement self-loops a sufficient number of times.[3]

In order to simulate moves of the form $a \to q$ (Figure 4 (b)), S_A simply decrements/increments the counter, depending on the sign of $|a|$, a number of times equal to the absolute value of $|a|$. The condition $y = 0$ ensures that S_A accepts only with the empty stack. The universal dotted branch in Figure 4 (c) is used to test that $y_m \leq e_m$ for some $1 \leq m \leq n$. A similar test for $y_p \geq l_p$ can be issued by replacing $y' = y+1$ with $y' = y-1$ on the loop. The following lemma is a concretisation of the above considerations:

Lemma 5 *Let $A = (Q, \Delta, F)$ be a TASC over a sized alphabet $(\Sigma, |.|)$ and let S_A be its corresponding APDS.*

1. *For any tree $t \in T(\Sigma)$ and any run $\pi : dom(t) \to Q$ of A on t, there exists an accepting run $\rho : \mathbb{N}^* \to (Q \times \Sigma \cup \Pi) \times \Gamma^*$ of S_A and an injective tree mapping $h : dom(t) \to dom(\rho)$ between π and ρ such that:*

$$\forall p \in dom(t)\ .\ \rho(h(p)) = \langle \pi(p), t(p), (|t_{|p}|)_1 \rangle \tag{1}$$

2. *For any accepting run $\rho : \mathbb{N}^* \to (Q \times \Sigma \cup \Pi) \times \Gamma^*$ of S_A, there exists a tree $t \in T(\Sigma)$, a run $\pi : dom(t) \to Q$ of A on t, and an injective tree mapping $h : dom(t) \to dom(\rho)$ between π and ρ satisfying (1).*

Proof (Part 1.) Let $t \in T(\Sigma)$ be a tree and $\pi : dom(t) \to Q$ be a run of A on t. We prove the existence of ρ and h by induction on the structure of t.

If $t = a$, the only runs of A on t are generated by applying rules of the form $a \to q$. In this case, for any $a \to q$, ρ is the accepting run of S_A starting in $\langle q, a, (|a|)_1 \rangle$ and ending with the empty stack as shown in Figure 4 (b). The tree mapping h is such that $h(\lambda) = \lambda$ and h is undefined everywhere else. Clearly, h is an injective tree mapping (cf. Definition 1), and property (1) is satisfied.

If $t = f(t_1,\ldots,t_n)$, a run of A over t has the form $t \xrightarrow{*} f(q_1,\ldots,q_n) \xrightarrow{\varphi} q$, for some runs $t_i \xrightarrow{*} q_i$, $1 \leq i \leq n$, and a transition rule $f(q_1,\ldots,q_n) \xrightarrow{\varphi} q \in \Delta$. Let $1 \leq r \leq n$ be the unique integer such that $|t| = |t_r| + c_r$, and $\models (\varphi \wedge \delta_r)(|t_1|,\ldots,|t_n|)$.

[3] Notice that since APDS do not have input, the universal branches are not synchronised, hence the iterations can be performed separately.

By Lemma 4, there exists a permutation $I \in Perm(n)$, and integers $d_k, e_m, l_p \in \mathbb{Z}$ such that :

$$\bigwedge_{k=1}^{n-1} |t_{I(k)}| - |t_{I(k+1)}| \diamond_k d_k \ \wedge \bigwedge_{m \in M \subseteq \{1,\ldots,n\}} x_m \leq e_m \ \wedge \bigwedge_{p \in P \subseteq \{1,\ldots,n\}} x_p \geq l_p$$

By the induction hypothesis, for each $1 \leq i \leq n$, S_A has an accepting run ρ_i starting in a configuration $\langle q_i, t_i(\lambda), (|t_i|)_1 \rangle$, and there exist injective tree mappings $h_i : dom(t_i) \rightarrow dom(\rho_i)$ satisfying property (1).

According to the construction in Figure 4 (a), S_A has a run θ starting in $\langle q, f, (|t|)_1 \rangle$ whose frontier forms a sequence $\mathbf{p} = \langle p_1, \ldots, p_n \rangle$ such that $\theta(p_k) = \langle q_{I(k)}, t_{I(k)}(\lambda), (|t_{I(k)}|)_1 \rangle$ for all $1 \leq k \leq n$.

Note that for each subterm t_i of the term $t = f(t_1, \ldots, t_n)$, $1 \leq i \leq n$, we can match the control state $\langle q_i, t_i(\lambda) \rangle$, from which the run ρ_i accepting t_i starts, with the control state $\langle q_{I(k)}, t_{I(k)}(\lambda) \rangle$ at the position $k = I^{-1}(i)$ of the frontier of θ. This is due to the construction in Figure 4 (a), which produces, for each transition rule $f(q_1, \ldots, q_n) \rightarrow q$ of A, a set of runs of S_A ending in control states of the form $\langle q_i, g \rangle$ for all $g \in \Sigma$. It is sufficient to choose from this set the run(s) for which $g_i = t_i(\lambda)$ for all $1 \leq i \leq n$.

Also, the construction in Figure 4 (a), when started with the value of the counter y being $|t|$, produces the values $|t_i|$ in the counters $y_{I^{-1}(i)}$, $1 \leq i \leq n$, such that $|t| = |t_r| + c_r$ and $\models (\varphi \wedge \delta_r)(|t_1|, \ldots, |t_n|)$. With the above considerations, this ensures that $\theta(p_k) = \rho_{I(k)}(\lambda)$ for all $1 \leq k \leq n$.

With these definitions, the accepting run ρ of S_A can be constructed as $\rho = \theta \bullet_{\mathbf{p}} \langle \rho_{I(1)}(\lambda), \ldots, \rho_{I(n)}(\lambda) \rangle$. One can see that ρ is accepting since each ρ_i is accepting for all $1 \leq i \leq n$.

The mapping h is defined such that $h(\lambda) = \lambda$, and for each $1 \leq i \leq n$, for all $p \in dom(t_i)$, $h(ip) = p_{I^{-1}(i)} \cdot h_i(p)$. The proof that h is an injective tree mapping (cf. Definition 1) satisfying property (1) is straightforward.

(Part 2.) Let $\rho : \mathbb{N}^* \rightarrow (Q \times \Sigma \cup \Pi) \times \Gamma^*$ be an accepting run of S_A. For an arbitrary position $p \in dom(\rho)$, let us denote by $\rho{\downarrow_p}$ the restriction of ρ to the set $\{u \in \mathbb{N}^* \mid \forall w \,.\, p \prec w \prec p \cdot u \Rightarrow \rho(w) \in \Pi \times \Gamma^*\}$. Let $P = \{p \in dom(\rho) \mid \rho(p) \in Q \times \Sigma \times \Gamma^*\}$, and notice that ρ can be written as a composition of *elementary runs* $\rho{\downarrow_p}$ for $p \in P$. We prove the existence of t, π, and h by induction on the number N of elementary runs in ρ.

For the base case $N = 1$, since ρ is accepting, the only possibility is that ρ is the result of simulating an existing rule $a \rightarrow q \in \Delta$ according to Figure 4 (b). Then, $\rho(\lambda) = \langle q, a, \gamma \rangle$, and by the construction of S_A, we have that $\gamma = (|a|)_1$. We then define $dom(t) = dom(\pi) = \{\lambda\}$, $t(\lambda) = a$, and $\pi(\lambda) = q$. Also, let $h(\lambda) = \lambda$, and h be undefined everywhere else. The proof that t, π, and h satisfy property (1) is straightforward.

For the induction step $N > 1$, let $\rho{\downarrow_\lambda}$ be the top-most elementary run of ρ. Let $\rho(\lambda) = \langle q, f, u \rangle$ be the starting configuration of ρ, and $\mathbf{p} = \langle p_1, \ldots, p_n \rangle$ be the sequence of positions on $fr(\rho{\downarrow_\lambda})$, i.e., $\rho = (\rho{\downarrow_\lambda}) \bullet_{\mathbf{p}} \langle \rho_{|p_1}, \ldots, \rho_{|p_n} \rangle$. Let $\rho(p_k) = \langle q_k, f_k, u_k \rangle$ for all $1 \leq k \leq n$.

By the induction hypothesis, for each $\rho_{|p_k}$, $1 \leq k \leq n$, there exist trees t_k, runs $\pi_k : dom(t_k) \to Q$ of the form $t_k \xrightarrow{*} q_k$, and injective tree mappings $h_k : dom(t_k) \to dom(\rho_k)$ satisfying property (1). Consequently, $f_k = t_k(\lambda)$ and $u_k = (|t_k|)_1$ for all $1 \leq k \leq n$.

Henceforth, we consider that $n > 1$, the case $n = 1$ being left to the reader. By the construction in Figure 4 (a), there exist:

- a transition rule $f(q_1, \ldots, q_n) \xrightarrow{\varphi(x_1, \ldots, x_n)} q \in \Delta$,
- a permutation $I \in Perm(n)$ such that $\models \varphi(|t_{I^{-1}(1)}|, \ldots, |t_{I^{-1}(n)}|)$,
- a reference position $1 \leq r \leq n$ such that $u = (|f(t_{I^{-1}(1)}, \ldots, t_{I^{-1}(n)})|)_1$, $|f(t_{I^{-1}(1)}, \ldots, t_{I^{-1}(n)})| = |t_{I^{-1}(r)}| + c_r$, and $\models \delta_r(|t_{I^{-1}(1)}|, \ldots, |t_{I^{-1}(n)}|)$.

With these definitions, let $t = f(t_{I^{-1}(1)}, \ldots, t_{I^{-1}(n)})$, and π be the tree defined as $\pi(\lambda) = q$, and, for all $1 \leq i \leq n$ and all $p \in dom(\pi_{I^{-1}(i)})$, $\pi(ip) = \pi_{I^{-1}(i)}(p)$. It is easy to see that π is a run of A over t.

The mapping h is defined such that $h(\lambda) = \lambda$, and for each $1 \leq i \leq n$, for all $p \in dom(t_{I^{-1}(i)})$, $h(ip) = p_{I^{-1}(i)} \cdot h_{I^{-1}(i)}(p)$. The proof that h is an injective tree mapping satisfying property (1) is straightforward. □

We can now formalise the main result of this subsection.

Theorem 2 *Let A be a TASC. The problem whether $L(A) = \emptyset$ is decidable.*

Proof Due to Lemma 5, we know that a tree with a root symbol $f \in \Sigma$ is accepted at a state q of a TASC $A = (Q, \Delta, F)$ over a sized alphabet Σ iff there is an accepting run from the control state $\langle q, f \rangle$ in the appropriate APDS $S_A = (Q_A, \Gamma, \delta_A, F_A)$. It is thus enough to use the result of [4] (mentioned at the begining of the section) to check whether for some $\langle q, f \rangle \in Q_A$ where $q \in F$, $pre^*_{\langle q,f \rangle}(\{\langle q_{fin}, \varepsilon \rangle\})$ is non-empty. Here, q_{fin} is the unique final state of the APDS S_A constructed according to Figure 4. □

Remark. The decidability of the emptiness problem for TASC can also be proved via a reduction to the class of *tree automata with one memory* [8] by encoding the size of a tree as a unary term. The inequality constraints from the guards of the TASC can be simulated analogously by adding increment/decrement self loops to the tree automata with one memory.

5 Semantics of Tree Updates

As explained in Section 2, there are three types of operations that commonly appear in procedures used for balancing binary trees after an insertion or deletion: (1) navigation in a tree, i.e., testing or changing the position that a pointer variable is pointing to in the tree, (2) testing or changing certain data fields of the encountered tree nodes, such as the colour of a node in a red-black tree, and (3) tree rotations. In addition, one has to consider the physical insertion or deletion to/from a suitable position in the tree as an input for the re-balancing.

It turns out that the TASC defined in Section 3 are not closed with respect to the effect of some of the above operations, in particular the ones that change the balance of subtrees (the difference between the size of the left and right subtree at a given position in the tree). Therefore, we now introduce a subclass of TASC called *restricted TASC* (rTASC), which we show to be effectively closed with respect to all the needed operations on balanced trees. Moreover, rTASC are closed with respect to intersection and union, amenable to determinisation and minimisation, though not closed with respect to complementation. The idea is to use rTASC to express loop invariants and pre- and post-conditions of programs as well as to perform the necessary reachability computations. TASC are then used in the associated language inclusion checks (where they arise via negation of rTASC).

Remark. To simplify the presentation of the effect of program statements on a set of memory configurations given by an rTASC, we suppose in the following that the statements do not lead to a memory error (like a null pointer dereference or similar). However, it is easy to implement tests for these potential errors over sets of memory configurations described by rTASCs in the same way as regular program conditions (i.e., if statements) are implemented, which we explain in Section 5.4.

5.1 Restricted TASC

A *restricted alphabet* is a sized alphabet consisting only of nullary and binary symbols and a size function of the form $|f(t_1,t_2)| = \max(|t_1|,|t_2|)+a$ with $a \in \mathbb{Z}$ for binary symbols. A *restricted* TASC is a TASC with a restricted alphabet and with binary rules of the form $f(q_1,q_2) \xrightarrow{|1|-|2|=b} q$ with $b \in \mathbb{Z}$ only.

Notice that any conjunction of guards of an rTASC and their negations reduces either to false, or to only one formula of the same form, i.e., $|1|-|2|=b$. Using this fact, one can show that the intersection of two rTASCs is again an rTASC, and that applying the determinisation of Section 4.1 to an rTASC yields another rTASC. Moreover, due to the fact that the guards of the transition rules of rTASCs contain at most two variables, it is not necessary to apply the potentially expensive step of converting them into the normal form described in Lemma 4, when deciding emptiness of rTASCs. Further, the intersection of an rTASC with a classical tree automaton is again an rTASC.[4] On the other hand, it is clear that rTASCs are not closed under complementation as inequality guards are not allowed.

Minimisation of rTASC. The simple form of the guards allows us to have a practical minimisation procedure based on the minimisation for classical bottom-up tree automata [9]. If $(\Sigma,|.|)$ is a restricted alphabet, let Σ_δ be the infinite ranked alphabet $\{\langle f,d\rangle \mid f \in \Sigma, d \in \mathbb{Z}\}$ with $\#(\langle f,d\rangle) = \#(f)$. For any $t \in T(\Sigma)$, let $\delta(t) \in T(\Sigma_\delta)$ be the tree defined as follows:

- $dom(t) = dom(\delta(t))$,
- for all $p \in dom(t)$, if $\#(t(p)) = 0$, we have $\delta(t)(p) = \langle t(p), |t(p)|\rangle$, and
- for all $p \in dom(t)$, if $\#(t(p)) = 2$, we have $\delta(t)(p) = \langle t(p), |t_{|p1}| - |t_{|p2}|\rangle$.

[4] A bottom-up tree automaton can be seen as a TASC in which all guards are true.

In other words, we record the balance of each subtree in the symbol that labels the root of the subtree. For constant symbols, we simply put their measures as labels in the tree. Obviously, δ is a (bijective) function from $T(\Sigma)$ to $T(\Sigma_\delta)$, which we extend pointwise to sets of trees. If A is an rTASC over the restricted alphabet $(\Sigma, |.|)$, let A_δ be the bottom-up tree automaton over Σ_δ defined by replacing each transition rule of A of the form:

- $a \to q$ by $\langle a, |a| \rangle \to q$, and
- $f(q_1, q_2) \xrightarrow{|1|-|2|=b} q$ by $\langle f, b \rangle (q_1, q_2) \to q$.

Note that we can always define A_δ over a finite subset of Σ_δ since the number of rules in A is finite. Moreover, the size of A (number of states) equals the size of A_δ. Last, the transformation of A into A_δ is always reversible.

Lemma 6 *Given an rTASC A over a sized alphabet $(\Sigma, |.|)$, for all trees $t \in T(\Sigma)$, we have $t \in \mathscr{L}(A)$ if and only if $\delta(t) \in \mathscr{L}(A_\delta)$.*

Proof We prove that $t \xrightarrow[A]{*} q$ iff $\delta(t) \xrightarrow[A_\delta]{*} q$ by induction on the structure of t. If $t = a \in \Sigma_0$, $a \xrightarrow[A]{} q$ if and only if $\delta(a) = \langle a, |a| \rangle \xrightarrow[A_\delta]{} q$. Otherwise, let $t = f(t_1, t_2) \xrightarrow[A]{*} f(q_1, q_2) \xrightarrow[A]{|1|-|2|=b} q$ with $t_i \xrightarrow[A]{*} q_i$, $1 \leq i \leq 2$. Then, $|t_1| - |t_2| = b$, hence $\delta(t) = \langle f, b \rangle (\delta(t_1), \delta(t_2))$. By the induction hypothesis, we have $\delta(t_i) \xrightarrow[A_\delta]{*} q_i$, and, by the definition of A_δ, $\langle f, b \rangle (q_1, q_2) \xrightarrow[A_\delta]{} q$. The other direction is symmetrical. □

Now, given an rTASC A, we compute A_δ, determinise and minimise it using the classical construction from [9], obtaining ${A_\delta}^{min}$. The minimal rTASC A^{min} is subsequently obtained by performing a reverse of the conversion from rTASC to tree automata on ${A_\delta}^{min}$, i.e., by moving back the integer constants from the symbols to the guards. To convince ourselves that A^{min} is indeed minimal, suppose there exists a smaller rTASC A' recognising the same language, i.e., $\mathscr{L}(A) = \mathscr{L}(A^{min}) = \mathscr{L}(A')$. Then, $\delta(\mathscr{L}(A)) = \delta(\mathscr{L}(A')) = \mathscr{L}(A'_\delta) = \mathscr{L}({A_\delta}^{min})$. Since A' and A'_δ have the same number of states, we contradict the minimality of ${A_\delta}^{min}$.

5.2 Representing Sets of Memory Configurations

To be able to describe how tree rotations (and the other considered operations) can be implemented over rTASC, we first have to explain how rTASC can be used for describing sets of memory configurations of programs manipulating balanced tree structures like red-black trees or AVL trees. Intuitively, we map memory configurations (i.e., heap graphs) having the form of trees node-by-node onto the trees accepted by rTASC, with the nodes labelled by (1) the variables pointing to them and by (2) the data elements stored in them. We also use the label *null* to denote null successors of leaf nodes.

Formally, let us consider a finite set of *pointer variables* $\mathcal{V} = \{x, y, \ldots\}$ and a finite set of data values $\mathcal{D}$, e.g., $\mathcal{D} = \{red, black\}$. In the following, we let $\Sigma = \mathcal{P}(\mathcal{V}) \times \mathcal{D} \cup \{null\}$. The arity function is defined as follows: $\#(f) = 2$ for all $f \in \mathcal{P}(\mathcal{V}) \times \mathcal{D}$, and $\#(null) = 0$. For any non-null symbol $f \in \mathcal{P}(\mathcal{V}) \times \mathcal{D}$, let $v(f) \subseteq \mathcal{V}$ and $d(f) \in \mathcal{D}$ denote the variables pointing to the tree node labelled with f and the data value of this node, respectivelly, i.e., $f = (v(f), d(f))$. For a tree $t \in T(\Sigma)$ and a variable $x \in \mathcal{V}$, we say that a node $p \in dom(t)$ is *pointed to by* x if and only if $t(p) \neq null$ and $x \in v(t(p))$. If there is no node pointed by a variable $x \in \mathcal{V}$ in a tree $t \in T(\Sigma)$, i.e., $\forall p \in dom(t).\ t(p) \neq null \Rightarrow x \notin v(t(p))$, we assume x to be null.[5]

For the rest of the section, let $A = (Q, \Delta, F)$ be an rTASC over Σ. We say that A represents a *set of memory configurations* if and only if for each $t \in L(A)$ and each $x \in \mathcal{V}$, there is at most one $p \in dom(t)$ that is pointed to by x. This condition can be always enforced by intersecting any given rTASC by the rTASC $A' = (Q', \Delta', Q')$ where $Q' = \mathcal{P}(\mathcal{V})$, and $\Delta' = \{null \longrightarrow \emptyset\} \cup \{ f(v_1, v_2) \xrightarrow{\top} v \mid v = v(f) \cup v_1 \cup v_2 \wedge v(f) \cap v_1 = v(f) \cap v_2 = v_1 \cap v_2 = \emptyset\}$. Intuitively, A' remembers in its control locations all so-far encountered variables and ensures that no variable is encountered twice.

An example of an rTASC representing within the described encoding the set of memory configurations correponding to the invariant in the red-black tree insertion procedure can be found at the beginning of Section 6.

5.3 Computing the Effect of Tree Rotations

Let $x \in \mathcal{V}$ be a fixed variable, and $A = (Q, \Delta, F)$ be an rTASC. We now give a method for deriving an rTASC $A' = (Q', \Delta', F')$ describing the set of trees that are the result of a *left rotation* applied to trees from $L(A)$ at the node pointed to by x. The case of the right tree rotation is very similar and so we skip it here.[6] In the description, we will be referring to Figure 5 illustrating the problem.

Let $R_x(\Delta) = \{(r_1, r_2) \in \Delta \times \Delta \mid r_1 : f(q_1, q_2) \xrightarrow{\varphi_3} q_3 \wedge r_2 : g(q_4, q_3) \xrightarrow{\varphi_5} q_5 \wedge x \in v(g)\}$ be the set of all the pairs of automata rules from Δ that can yield a rotation, and be modified because of it. Other rules may then have to be modified to reflect:

- changes of some control states, for instance the change of q_5 to q_3' in rule r_3 from Figure 5, or
- changes of balance resulting from the rotation, i.e., changes in the difference between the sizes of left and right subtrees, which get propagated from the rotated subtree upwards.

To define the resulting automaton, we use an auxiliary set $D \subseteq \mathbb{Z}$ which contains all the *changes in balance* that may occur in any tree from $L(A)$, due to a rotation at x. The set D is the smallest set such that:

[5] For simplicity, we do not explicitly distinguish null and undefined pointer values. Such a distinction could, however, be easily introduced.

[6] In fact, it can be implemented by temporarily swapping the child nodes in the involved rules, doing a left rotation, and then swapping the child nodes again.

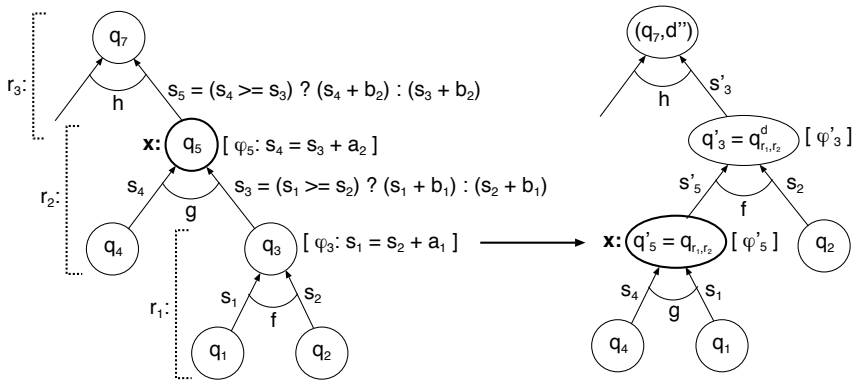

Fig. 5 Left rotation on an rTASC

- $iniD(r_1, r_2) \in D$, for all $(r_1, r_2) \in R_x$, and
- $leftD(a,d) \in D,\ rightD(a,d) \in D$, for all $d \in D$ and $f(q_1,q_2) \xrightarrow{|1| = |2| + a} q_3 \in \Delta$.

The function *iniD* computes the initial disbalance caused by the rotation, while $leftD$ and *rightD* propagate upward the disbalance that happened in the left or right subtree of some node, respectivelly. The definitions of these functions are the subject of Sections 5.3.1 and 5.3.2. Lemma 7 shows that the set D is finite, which guarantees that A' can be computed in a finite number of steps.

The set of states of A' is defined as $Q' = Q \cup Q_x \cup Q_x^D \cup Q^D$, where Q_x, Q_x^D and Q^D are pairwise disjoint sets, all disjoint from Q, defined as:

- $Q_x = \{q_{r_1,r_2} \mid (r_1,r_2) \in R_x(\Delta)\}$ contains a new state for each pair of transition rules involved in the rotation, which accepts the former root of the rotated subtree that went down in the rotation.
- $Q_x^D = \{q_x^d \mid q_x \in Q_x \wedge d \in D\}$ is the set of states accepting the nodes that went up in the rotation and became the new root of the rotated subtree.
- $Q^D = \{q^d \mid q \in Q \ \wedge \ d \in D\}$ is the set of states accepting all contexts of subtrees changed by the rotation (i.e., the tree nodes that appear above the rotated subtree).

The set Δ' of transition rules of A' is defined as the limit of the increasing sequence of sets $\Delta'_0 \subseteq \Delta'_1 \subseteq \Delta'_2, \ldots$, i.e., $\Delta' = \bigcup_{i \geq 0} \Delta'_i$, where $\Delta'_0 = \Delta$, and Δ'_{i+1} is obtained from Δ'_i by applying one of the rules in Figure 6. Since, by Lemma 7, the set D is finite, it is obvious that the limit is reached in a finite number of steps. The functions *lPhi* and *rPhi* that compute the guards of the newly added rules, are given in Section 5.3.1.

The set of final states of A' is defined as $F' = \{q^d \mid q \in F\} \cup \{ q^d_{r_1,r_2} \mid (r_1,r_2) \in R_x \wedge d = iniD(r_1,r_2) \wedge r_1 : f(q_1,q_2) \xrightarrow{\varphi_3} q_3 \wedge r_2 : g(q_4,q_3) \xrightarrow{\varphi_5} q_5 \wedge q_5 \in F\}$. Intuitivelly, the states from $\{q^d \mid q \in F\}$ ensure that we accept only the trees in which a rotation actually occurred in some subtree. Additionaly, the states $q^d_{r_1,r_2} \in$

$$\frac{(r_1,r_2)\in R_x(\Delta'_i)\quad r_1 : f(q_1,q_2)\xrightarrow{\varphi_3} q_3\quad r_2 : g(q_4,q_3)\xrightarrow{\varphi_5} q_5}{g(q_4,q_1)\xrightarrow{lPhi(r_1,r_2)} q_{r_1,r_2}\ \in\ \Delta'_{i+1}}\ R_{1a}$$

$$\frac{(r_1,r_2)\in R_x(\Delta'_i)\quad r_1 : f(q_1,q_2)\xrightarrow{\varphi_3} q_3\quad r_2 : g(q_4,q_3)\xrightarrow{\varphi_5} q_5}{f(q_{r_1,r_2},q_2)\xrightarrow{rPhi(r_1,r_2)} q_{r_1,r_2}^{iniD(r_1,r_2)}\ \in\ \Delta'_{i+1}}\ R_{1b}$$

$$\frac{(r_1,r_2)\in R_x(\Delta'_i)\quad r_2 : g(q_4,q_3)\xrightarrow{\varphi_5} q_5\quad h(q_5,q_6)\xrightarrow{|1|=|2|+a} q_7\ \in\ \Delta'_i}{h(q_{r_1,r_2}^{iniD(r_1,r_2)},q_6)\xrightarrow{|1|=|2|+a+iniD(r_1,r_2)} q_7^{leftD(a,iniD(r_1,r_2))}\ \in\ \Delta'_{i+1}}\ R_{2a}$$

$$\frac{(r_1,r_2)\in R_x(\Delta'_i)\quad r_2 : g(q_4,q_3)\xrightarrow{\varphi_5} q_5\quad h(q_6,q_5)\xrightarrow{|1|=|2|+a} q_7\ \in\ \Delta'_i}{h(q_6,q_{r_1,r_2}^{iniD(r_1,r_2)})\xrightarrow{|1|=|2|+a-iniD(r_1,r_2)} q_7^{rightD(a,iniD(r_1,r_2))}\ \in\ \Delta'_{i+1}}\ R_{2b}$$

$$\frac{f(q_1,q_2)\xrightarrow{|1|=|2|+a} q_3\ \in\ \Delta'_i\quad d\in D}{f(q_1^d,q_2)\xrightarrow{|1|=|2|+a+d} q_3^{leftD(a,d)}\ \in\ \Delta'_{i+1}}\ R_{3a}$$

$$\frac{f(q_1,q_2)\xrightarrow{|1|=|2|+a} q_3\ \in\ \Delta'_i\quad d\in D}{f(q_1,q_2^d)\xrightarrow{|1|=|2|+a-d} q_3^{rightD(a,d)}\ \in\ \Delta'_{i+1}}\ R_{3b}$$

Fig. 6 Rules for computing the effect of left rotations on rTASCs

F', accepting the root of the rotated subtree, become final if the rotation occurs at a state q_5 accepting the root node of the original tree (i.e., if $q_5 \in F$).

5.3.1 The Root of the Rotated Subtree: Computing lPhi, rPhi, and iniD

Let us consider $(r_1,r_2)\in R_x(\Delta)$ where $r_1 : f(q_1,q_2)\xrightarrow{\varphi_3} q_3$ and $r_2 : g(q_4,q_3)\xrightarrow{\varphi_5} q_5$, as in Figure 5. Suppose that $\varphi_3 : |1| = |2| + a_1$ and let us denote the sizes of the subtrees read at q_1 and q_2 before the rotation, by s_1 and s_2, respectively. Let the size function associated with f be $|f(t_1,t_2)| = \max(|t_1|,|t_2|) + b_1$, and let s_3 denote the size of the subtree labelled by q_3 before the rotation. Also, suppose that $\varphi_5 : |1| = |2| + a_2$ and let us denote the size of the sub-tree read at q_4 before the rotation as s_4. Finally, let the size function associated with g be $|g(t_1,t_2)| = \max(|t_1|,|t_2|) + b_2$, and let s_5 denote the size of the subtree labelled by q_5 before the rotation. We denote s'_5 and s'_3 the sizes obtained at q'_5 and q'_3 after the rotation.

The key observation that allows us to compute the guards $\varphi'_5 : lPhi(r_1, r_2)$ and $\varphi'_3 : rPhi(r_1, r_2)$ of the rules that accept the root of the rotated subtree, as well as the change in balance $d = iniD(r_1, r_2)$ caused by the rotation, is that due to the chosen form of guards and sizes, we can always compute any two of the sizes s_1, s_2, s_4 from the remaining one. Indeed,

- for $a_1 \geq 0$, we have $s_3 = s_1 + b_1 = s_2 + a_1 + b_1 = s_4 - a_2$, whereas
- for $a_1 < 0$, we have $s_3 = s_2 + b_1 = s_1 - a_1 + b_1 = s_4 - a_2$.

Computing φ'_3, φ'_5, and d is then just a complex exercise in case splitting. Notice that all the cases can be distinguished statically according to the mutual relations of the constants a_1, b_1, a_2, and b_2. In the case of φ'_5, we obtain the following:

1. For $a_1 \geq 0$, we have $s_4 = s_1 + b_1 + a_2$, and so $\varphi'_5 : |1| = |2| + b_1 + a_2$.
2. For $a_1 < 0$, we have $s_4 = s_1 - a_1 + b_1 + a_2$, and so $\varphi'_5 : |1| = |2| - a_1 + b_1 + a_2$.

The guard φ'_3 is a bit more complex. We distinguish two cases: $\Phi_{4\geq 1} : s_4 \geq s_1$ and $\Phi_{4<1} : s_4 < s_1$. Now we rewrite the conditions $s_4 \geq s_1$ and $s_4 < s_1$ using the relation between s_4 and s_1 described above for $a_1 \geq 0$ and $a_1 < 0$:

1. $\Phi_{4\geq 1} : s_4 \geq s_1 \iff (a_1 \geq 0 \wedge b_1 + a_2 \geq 0) \vee (a_1 < 0 \wedge -a_1 + b_1 + a_2 \geq 0)$. If $\Phi_{4\geq 1}$ holds, then $s'_5 = s_4 + b_2$. Further, we distinguish between the following cases:
   (a) For $a_1 \geq 0 \wedge b_1 + a_2 \geq 0$, we get $s'_5 = s_1 + b_1 + a_2 + b_2$ (as $a_1 \geq 0$), i.e., $s_1 = s'_5 - b_1 - a_2 - b_2$. Taking into account that $s_1 = s_2 + a_1$, we obtain $\varphi'_3 : |1| = |2| + a_1 + b_1 + a_2 + b_2$.
   (b) For $a_1 < 0 \wedge -a_1 + b_1 + a_2 \geq 0$, we have $s'_5 = s_1 - a_1 + b_1 + a_2 + b_2$ (as $a_1 < 0$), i.e., $s_1 = s'_5 + a_1 - b_1 - a_2 - b_2$. Using that $s_1 = s_2 + a_1$, we obtain $\varphi'_3 : |1| = |2| + b_1 + a_2 + b_2$.
2. $\Phi_{4<1} : s_4 < s_1 \iff (a_1 \geq 0 \wedge b_1 + a_2 < 0) \vee (a_1 < 0 \wedge -a_1 + b_1 + a_2 < 0)$. If $\Phi_{4<1}$ holds, we have $s'_5 = s_1 + b_2$, and so $\varphi'_3 : |1| = |2| + a_1 + b_2$.

The computation of the change in the balance d is similar to the above. The first case to be considered is $\Phi_{4\geq 3} : s_4 \geq s_3 \iff a_2 \geq 0$. Here, $s_5 = s_4 + b_2$. To compute the change in the sizes reached at q_5 and q'_3, which is to be compensated in the transitions to come after q'_3 instead of q_5, we need to compute s'_3 as a function of s_4 (then, in the difference, s_4 will be eliminated). We can write the following:

$$s'_3 = \begin{cases} \text{if } \Phi_{4\geq 1} : \\ \quad \begin{cases} \text{if } s_4 + b_2 \geq s_2 : s_4 + b_2 + b_1 \\ \text{if } s_4 + b_2 < s_2 : s_2 + b_1 \end{cases} \\ \text{if } \Phi_{4<1} : \\ \quad \begin{cases} \text{if } s_1 + b_2 \geq s_2 : s_1 + b_2 + b_1 \\ \text{if } s_1 + b_2 < s2 : s_2 + b_1 \end{cases} \end{cases}$$

Let us first consider the subcase when $\Phi_{4\geq 1}$. It has two further subcases $s_4 + b_2 \geq s_2$ and $s_4 + b_2 < s_2$, which we can again rewrite by using the known relations between s_4 and s_2 for $a_1 \geq 0$ ($s_2 + a_1 + b_1 = s_4 - a_2$) and $a_1 < 0$ ($s_2 + b_1 = s_4 - a_2$). We get:

1. $s_4 + b_2 \geq s_2 \iff (a_1 \geq 0 \wedge a_1 + b_1 + a_2 + b_2 \geq 0) \vee (a_1 < 0 \wedge b_1 + a_2 + b_2 \geq 0)$. In this case, we have $s'_3 = s_4 + b_2 + b_1$, and so $d = b_1$.

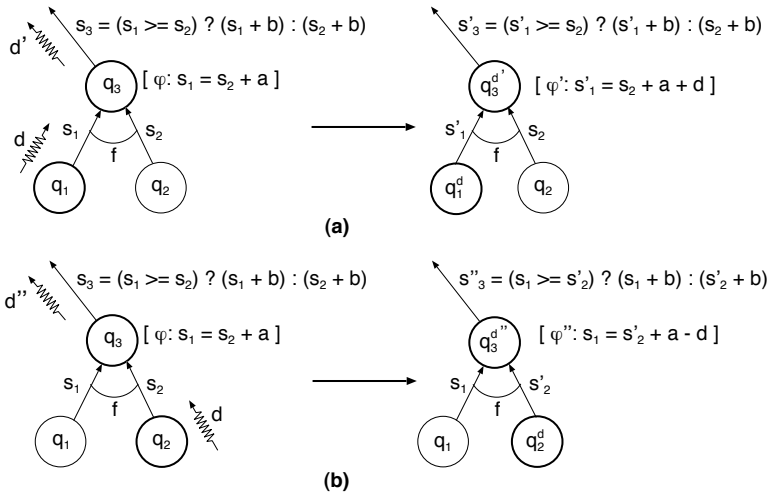

Fig. 7 Propagation of changes in balance in an rTASC

2. $s_4+b_2<s_2 \iff (a_1\geq 0 \wedge a_1+b_1+a_2+b_2<0) \vee (a_1<0 \wedge b_1+a_2+b_2<0)$. Here, $s'_3=s_2+b_1$, and we distinguish the following subcases:
   (a) for $a_1\geq 0 \wedge a_1+b_1+a_2+b_2<0$, $s'_3=s_2+b_1=s_4-a_1-b_1-a_2+b_1=s_4-a_1-a_2$, and so $d=-a_1-a_2-b_2$.
   (b) for $a_1<0 \wedge b_1+a_2+b_2<0$, $s'_3=s_2+b_1=s_4-b_1-a_2+b_1=s_4-a_2$, and so $d=-a_2-b_2$.

   The remaining cases of the computation of d are similar to the above.

5.3.2 Propagating Changes in Balance through rTASC: Computing leftD, rightD

We now study the way how a change in balance caused by a rotation is propagated from the subtree where the rotation took place to the root of the entire tree in which the rotation happened. The propagation is a part of the rules $(R2a)$–$(R3b)$ from Figure 6, and it is illustrated in Figure 7 on a rule $f(q_1,q_2) \xrightarrow{\varphi} q_3$ whose left or right child size changes by a value $d \in D$. Consequently, rules of the form $f(q_1^d,q_2) \xrightarrow{\varphi'} q_3^{d'}$ or $f(q_1,q_2^d) \xrightarrow{\varphi''} q_3^{d''}$ are generated by the rules $(R2a)$–$(R3b)$, depending on whether the change in balance originates from the left or the right. Since we consider just one rotation in every tree (at a given node pointed to by the pointer variable x), the change can never come from both sides. The guards of the new rules, compensating the change in balance that happens between the child nodes, are $\varphi' : |1| = |2| + a + d$ or $\varphi'' : |1| = |2| + a - d$, respectivelly. It remains to analyse the changes in the balance that are propagated upwards after d comes from the bottom, i.e., the way the values $d' = leftD(a,d)$ or $d'' = rightD(a,d)$ are computed.

Suppose the change in balance is coming from the left as in Figure 7 (a). We distinguish the cases of $a \geq 0$ and $a < 0$. (1) For $a \geq 0$, the original size at q_3 is

$s_3 = s_1 + b$ where s_1 is the original size at q_1. After the change d happens at q_1, i.e., $s'_1 - s_1 = d$, we have the following subcases: (1.1) For $a+d \geq 0$, we have $s'_3 = s'_1 + b$, i.e., $d' = d$, and so we have the same change in the size at q_3 as at q_1. (1.2) For $a+d < 0$, we have $s'_3 = s_2 + b = s_1 - a + b$, and hence $d' = -a$. (2) For $a < 0$, $s_3 = s_2 + b$. In this case, (2.1) for $a+d \geq 0$, $s'_3 = s'_1 + b = s_1 + d + b = s_2 + a + d + b$, and so $d' = a + d$, and (2.2) for $a+d < 0$, $s'_3 = s_2 + b$, and thus $d' = 0$. To summarize:

$$leftD(a,d) = \begin{cases} d & \text{if } a \geq 0 \text{ and } a+d \geq 0, \\ -a & \text{if } a \geq 0 \text{ and } a+d < 0, \\ a+d & \text{if } a < 0 \text{ and } a+d \geq 0, \\ 0 & \text{if } a < 0 \text{ and } a+d < 0 \end{cases}$$

Similarly, when the change is coming from the right as in Figure 7 (b), we have the following cases: (1) For $a \geq 0$, the original size at q_3 is $s_3 = s_1 + b$, and we have the following subcases for the new size: (1.1) For $a - d \geq 0$, $s''_3 = s_1 + b$, and so $d'' = 0$. (1.2) For $a - d < 0$, $s''_3 = s'_2 + b = s_2 + d + b = s_1 - a + d + b$, and thus $d'' = -a + d$. (2) For $a < 0$, $s_3 = s_2 + b$. Further, (2.1) for $a - d \geq 0$, $s''_3 = s_1 + b = s_2 + a + b$, i.e., $d'' = a$, and (2.2) for $a - d < 0$, $s''_3 = s'_2 + b = s_2 + d + b$, and hence $d'' = d$. To summarize:

$$rightD(a,d) = \begin{cases} 0 & \text{if } a \geq 0 \text{ and } a-d \geq 0, \\ -a+d & \text{if } a \geq 0 \text{ and } a-d < 0, \\ a & \text{if } a < 0 \text{ and } a-d \geq 0, \\ d & \text{if } a < 0 \text{ and } a-d < 0 \end{cases}$$

We can now close our construction by showing that the set D of possible changes in the sizes of the trees being handled is finite, which guarantees termination of the algorithm for computing the rTASC describing the effect of a tree rotation on trees from an rTASC-described set.

Lemma 7 *For an rTASC $A = (Q, \Delta, F)$ over a set of variables $\mathcal{V}$ and a variable $x \in \mathcal{V}$, the set D of the possible changes in balance of subtrees of the trees generated by a left tree rotation at a node pointed by x in the trees from $L(A)$ is finite.*

Proof Let $M = \max\big(\{|iniD(r_1,r_2)| \mid (r_1,r_2) \in R_x(\Delta)\} \cup \{a \mid (f(q_1,q_2) \xrightarrow{|1|=|2|+a} q) \in \Delta\}\big)$. Notice that D is the limit of an increasing sequence $D_0 \subseteq D_1 \subseteq D_2 \ldots$, where $D_0 = \{iniD(r_1,r_2) \mid (r_1,r_2) \in R_x(\Delta)\}$, and for each $i \geq 0$ there exists some rule $f(q_1,q_2) \xrightarrow{|1|=|2|+a} q$ and some $d \in D_i$ such that either $D_{i+1} = D_i \cup \{leftD(a,d)\}$, or $D_{i+1} = D_i \cup \{rightD(a,d)\}$.

By fixpoint induction, we show that $D \subseteq [-M,M]$, which implies that D is finite. $D_0 \subseteq [-M,M]$ by the choice of M. If $D_{i+1} = D_i \cup \{leftD(a,d)\}$, it is enough to show that $-M \leq leftD(a,d) \leq M$, if $-M \leq a,d \leq M$. The most interesting case is when $leftD(a,d) = a+d$, for $a < 0$ and $a+d \geq 0$. In this case $a \leq a+d \leq d$, therefore $-M \leq a+d \leq M$. The proof that $-M \leq rightD(a,d) \leq M$, for $-M \leq a,d \leq M$, is similar. □

Notice that it can be shown that Lemma 7 does not hold for general TASCs, due to the fact that the computation of the set D might diverge, in the general case.

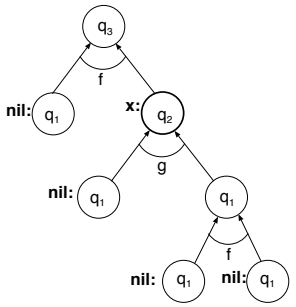

Fig. 8 Testing pointers

5.4 Other Operations on Sets of Trees Described by rTASC

Let us now briefly show that, in addition to the tree rotations, rTASC are also closed with respect to all other operations that we commonly need when dealing with balanced binary trees. We have listed these operations in Section 2. We are only giving an informal description of the algorithms for performing these operations over rTASC here—their formalisation is, however, straightforward.

Testing and Changing Pointers and Data. We first consider the operation of testing whether two pointer expressions refer to the same node of a tree. Examples of such tests are expressions `x == root` or `x->parent->right == x`. In general, we consider any test of the form e_1==e_2 where e_1, e_2 are of the form v->n_1->n_2->...n_m with $v \in \mathscr{V}$, $m \in \mathbb{N}$, and $n_1,...,n_m \in \{\texttt{left},\texttt{right},\texttt{parent}\}$. Suppose we are given an rTASC A recognising a set S of trees and a pointer equality test c. The rTASC describing the subset S' of S of the trees that meet c is the intersection of A and a TASC A_c encoding c. Since c describes a regular set of trees, this TASC can be easily derived in an algorithmic way.

To illustrate the construction, let us present an example of A_c for the condition `x->parent->right == x`. Recall that $\Sigma = \mathscr{P}(\mathscr{V}) \times \mathscr{D} \cup \{null\}$. Then, $A_c = (Q, \Delta, F)$ is defined by $Q = \{q_1, q_2, q_3\}$, $F = \{q_3\}$, and $\Delta = \{null \to q_1\} \cup \{f(q_1,q_1) \to q_1, g(q_1,q_1) \to q_2, f(q_1,q_2) \to q_3, f(q_3,q_1) \to q_3, f(q_1,q_3) \to q_3 \mid f,g \in \mathscr{P}(\mathscr{V}) \times \mathscr{D}, x \notin v(f), x \in v(g)\}$. Here, the pointer referencing pattern gets simply captured in the rule $f(q_1,q_2) \to q_3$. An example run of the automaton is illustrated in Figure 8.

Second, pointer assignments of the form v' = v->n_1-> n_2->...n_m can be implemented in our framework as a simple transformation of the input rTASC that removes v' from the node where it is in the input tree and adds it to the node referenced by v->n_1->n_2->...n_m. Note that we do not treat assignments of the form v->n_1->n_2->...n_m = v'->n'_1->n'_2->...$n'_{m'}$, i.e., destructive updates. We hide these assignments by encoding the effect of the entire procedures in which they appear, i.e., rotations and physical insertion or deletion of nodes. These operations temporarily break the tree shape of the structures being handled by introducing pointer sharing and even cycles. We suppose the correctness of these operations to be checked independently. A generalisation of our method to be able to handle

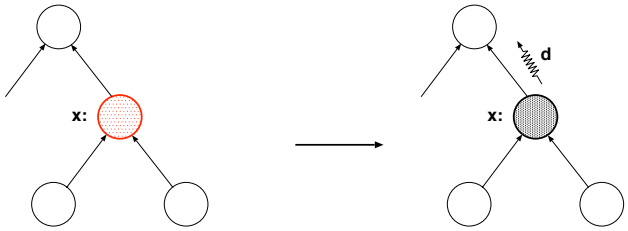

Fig. 9 Changing data contents in an rTASC

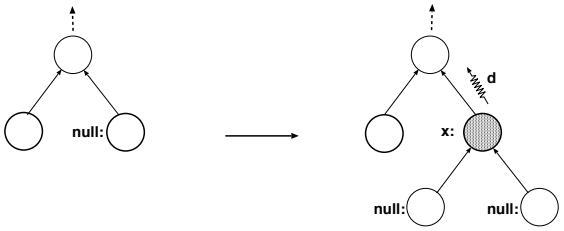

Fig. 10 Inserting a node in an rTASC

even the internal implementation of these procedures is an interesting subject for further research.

Testing and changing the data contents of the nodes pointed to by some pointer expression of the form $v\texttt{->}n_1\texttt{->}n_2\texttt{->}...n_m$ is an analogy of the pointer reference checking and pointer assignments. However, changing the data contents of some node (e.g., recolouring of some node in a red-black tree), can change the size of the appropriate subtree. In this case, the guards of all the transition rules that can be fired above the node that is recoloured (see Figure 9 for an example assignement $x\texttt{->}colour = black$) are to be changed in the same way as in Section 5.3.2 in order to reflect the change d in the balance that happens at the recoloured node.

Inserting New Nodes. Next, concerning the physical insertion of a new leaf node, recall that we suppose the null successors of such memory nodes to be explicitly represented by `null`-labelled nodes in our model. Compared to the real content of the memory, we thus add one layer of nodes (as null nodes are not allocated in the real memory). Inserting a new leaf memory node pointed to by a pointer variable x (which is undefined or null before) and having a data value c then amounts to replacing one of the null sons of some node by a new, non-null node with two null sons. We abstract here the sortedness property and we just pick randomly the place to insert the new leaf. The operation can be implemented as a simple transformation that modifies the input rTASC by non-deterministically choosing some null node, recolouring it to $(\{x\},c)$, and adding two null sons to it. Then, the changes in the number of nodes marked by c have to be propagated using the same technique as explained in Section 5.3.2 (see Figure 10 for an illustration).

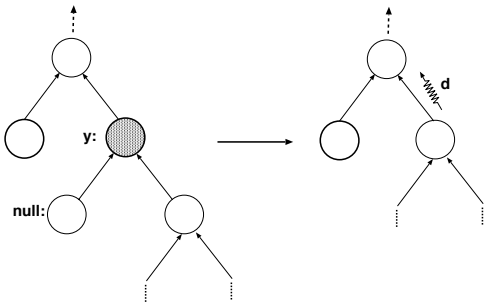

Fig. 11 Deleting a node in an rTASC

Deleting Nodes. Finally, the deletion of a frontier node pointed to by some pointer variable y is modelled by removing the rules $(\{y\},c)(q,q_{\texttt{null}}) \xrightarrow{\varphi} q_y$, where $null \rightarrow q_{\texttt{null}}$ (note that a frontier node has at least one null son). In the remaining rules, we simply replace all the appearances of q_y by all the q states that appeared in the deleted rules. Subsequently, we use again the same technique as in Section 5.3.2 to handle the changes in the balance resulting from a deletion of a node. See Figure 11 for an illustration of deleting a frontier node with a null left successor.

6 A Case Study: The Red-Black Tree Insertion

To illustrate our methodology, we show how to prove an invariant for the main loop in the procedure RB-Insert. (Note that all the steps can be done *fully automatically*.) This invariant is needed to prove the correctness of the insertion procedure given in Section 2, that is, given a valid red-black tree as input to the procedure, the output is also a valid red-black tree. The invariant is the conjunction of the following facts:

1. x is pointing to a non-null node in the tree.
2. If a node is red, then (i) its left son is either black or pointed to by x, and (ii) its right son is either black or pointed to by x. This condition is needed as during the re-balancing of the tree, a red node can temporarily become a son of another red node.
3. The root is either black or x is pointing to the root.
4. If x is not pointing to the the root and points to a node whose father is red, then x points to a red node.
5. Each maximal path from the root to a leaf contains the same number of black nodes. This is the last condition from the definition of red-black trees from Section 2.

In this example, we have $\mathscr{V} = \{x\}$, $\mathscr{D} = \{red, black\}$, and thus $\Sigma = (\{\emptyset, \{x\}\} \times \{red, black\}) \cup \{null\}$. For presentation purposes, we denote the symbol $(\emptyset, c) \in \Sigma$ by c, and c_x stands for $(\{x\}, c) \in \Sigma$, where $c \in \{red, black\}$. Also, if no guard is specified on a binary rule, we assume it to be $|1| = |2|$. Let $R = \{null \rightarrow$

$q_b, red(q_b, q_b) \to q_r, black(q_{b/r}, q_{b/r}) \to q_b\}$. The loop invariant is given by the following rTASC A_1.

$$
\begin{array}{ll}
A_1 : F = \{q_{rx}, q_{bx}, q'_{bx}\},\ \Delta = R \cup & \\
\{black_x(q_{b/r}, q_{b/r}) \to q_{bx}\ (1), & black(q_{bx/rx}, q_{b/r}) \to q'_{bx}(2) \\
black(q'_{bx/rx}, q_{b/r}) \to q'_{bx}, & black(q_{b/r}, q'_{bx/rx}) \to q'_{bx}\ (3), \\
black(q_{b/r}, q'_{bx/rx}) \to q'_{bx}, & red_x(q_b, q_b) \to q_{rx}, \\
red(q'_{bx}, q_b) \to q'_{rx}, & red(q_b, q'_{bx}) \to q'_{rx}, \\
red(q_{rx}, q_b) \to q'_{rx}\ (4), & red(q_b, q_{rx}) \to q'_{rx}\ (5)\}
\end{array}
$$

Intuitively, q_b labels black nodes and q_r red nodes which do not have a node pointed to by x below them. q_{bx} and q_{rx} mean the same except that they label a node which is pointed to by x. Primed versions of q_{bx} and q_{rx} are used for nodes which have a subnode pointed to by x. In the following, this intuitive meaning of states will be changed by the program steps. We refer to the pseudo-code of Section 2.

If the loop entrance condition `x!= root && x->parent->color == red` is true, we obtain a new automaton A_2. It is given by modifying A_1 as follows: $F = \{q'_{bx}\}$ and the rules (1), (2), and (3) are removed.

If the condition `x->parent == x->parent->parent->left` is true, we take A_2, change rule (4) to $red(q_{rx}, q_b) \to q''_{rx}$, rule (5) to $red(q_b, q_{rx}) \to q''_{rx}$ and add a rule $black(q''_{rx}, q_{b/r}) \to q'_{bx}\ (6)$ to obtain A_3. Now, q''_{rx} accepts the father of the node pointed by `x` and q'_{rx} its grandfather.

If the condition `x->parent->parent->right->color == red` holds, we obtain the automaton A_4 that is like A_3 except for rule (6) changed into $black(q''_{rx}, q_r) \to q'_{bx}$.

The recolouring step `x->parent->color = black` changes some guards on rules and leads to a propagation of the change through the automaton. The result is A_5:

$$
\begin{array}{ll}
A_5 : F = \{q'_{bx}\}, \Delta = R\ \cup & \\
\{black(q'_{bx/rx}, q_{b/r}) \xrightarrow{|1| = |2|+1} q'_{bx}, & red_x(q_b, q_b) \to q_{rx}, \\
black(q_{b/r}, q'_{bx/rx}) \xrightarrow{|1|+1 = |2|} q'_{bx}, & red(q'_{bx}, q_b) \xrightarrow{|1| = |2|+1} q'_{rx}, \\
black(q''_{rx}, q_r) \xrightarrow{|1| = |2|+1} q'_{bx}\ (7), & red(q_b, q'_{bx}) \xrightarrow{|1|+1 = |2|} q'_{rx}, \\
black(q_{rx}, q_b) \to q''_{rx}, & black(q_b, q_{rx}) \to q''_{rx}\}
\end{array}
$$

After the recolouring step `x->parent->parent->right->color = black`, we get A_6 which is A_5 where we change rule (7) to $black(q''_{rx}, q_b) \to q'_{bx}$. Note that no propagation is needed in this case.

After the recolouring step `x->parent->parent->color = red`, which introduces changes on guards, and the propagation of these changes, we obtain:

$A_7 : F = \{q'_{bx}\}, \Delta = R \cup$
$\{black(q'_{bx/rx}, q_{b/r}) \to q'_{bx}, \quad black(q_{b/r}, q'_{bx/rx}) \to q'_{bx},$
$black(q_{rx}, q_b) \to q''_{rx}, \quad black(q_b, q_{rx}) \to q''_{rx},$
$red_x(q_b, q_b) \to q_{rx}\ (8), \quad red(q'_{bx}, q_b) \to q'_{rx},$
$red(q''_{rx}, q_r) \to q'_{bx}\ (9), \quad red(q_b, q'_{bx}) \to q'_{rx}\}$

After `x = x->parent->parent`, we get A_8 derived from A_7 by changing rule (8) to $red(q_b, q_b) \to q_{rx}$ and rule (9) to $red_x(q''_{rx}, q_b) \to q'_{bx}$.

This takes care of case 1 and one can then check that $\mathscr{L}(A_8) \subseteq \mathscr{L}(A_1)$.

For case 2, we have to go back to automaton A_3 and apply the fact that the conditional `x->parent->parent->right->color == red` is false, i.e., `x->parent->parent->right->color == black` must be true. The result is:

$A_9 : F = \{q'_{bx}\}, \Delta = R \cup$
$\{black(q'_{bx/rx}, q_{b/r}) \to q'_{bx}, \quad black(q_{b/r}, q'_{bx/rx}) \to q'_{bx},$
$black(q''_{rx}, q_b) \to q'_{bx}, \quad red_x(q_b, q_b) \to q_{rx}\ (11),$
$red(q'_{bx}, q_b) \to q'_{rx}, \quad red(q_b, q'_{bx}) \to q'_{rx},$
$red(q_b, q_{rx}) \to q''_{rx}\ (12), \quad red(q_{rx}, q_b) \to q''_{rx}\ (10)\}$

After the condition `x == x->parent->right`, A_9 is changed into A_{10} by removing rule (10). After `x = x->parent`, A_{10} is changed into A_{11} by changing rule (11) to $red(q_b, q_b) \to q_{rx}$ and rule (12) to $red_x(q_b, q_{rx}) \to q''_{rx}$.

Now the operation `Left-Rotate(T,x)` introduces new states and transitions and we get the TASC A_{12}. Notice that no rebalancing is necessary.

$A_{12} : F = \{q'_{bx}\}, \Delta = R \cup$
$\{black(q'_{bx/rx}, q_{b/r}) \to q'_{bx}, \quad black(q_{b/r}, q'_{bx/rx}) \to q'_{bx},$
$black(q_{rot2}, q_b) \to q'_{bx}, \quad red_x(q_b, q_b) \to q_{rot1},$
$red(q'_{bx}, q_b) \to q'_{rx}, \quad red(q_b, q'_{bx}) \to q'_{rx},$
$red(q_{rot1}, q_b) \to q_{rot2}\}$

After `x->parent->color = black` and a propagation of the changes in the balance, we obtain:

$A_{13} : F = \{q'_{bx}\}, \Delta = R \cup$
$\{black(q'_{bx/rx}, q_{b/r}) \xrightarrow{|1| = |2| + 1} q'_{bx}, \ red_x(q_b, q_b) \to q_{rot1},$
$black(q_{b/r}, q'_{bx/rx}) \xrightarrow{|1| + 1 = |2|} q'_{bx}, \quad red(q'_{bx}, q_b) \xrightarrow{|1| = |2| + 1} q'_{rx},$
$black(q_{rot2}, q_b) \xrightarrow{|1| = |2| + 1} q'_{bx}, \quad red(q_b, q'_{bx}) \xrightarrow{|1| + 1 = |2|} q'_{rx},$
$black(q_{rot1}, q_b) \to q_{rot2}\}$

After `x->parent->parent->color = red`, we obtain:

$$
\begin{array}{l}
A_{14}: F = \{q'_{bx}\}, \Delta = R \cup \\
\{black(q'_{bx/rx}, q_{b/r}) \to q'_{bx}, \qquad red_x(q_b, q_b) \to q_{rot1}, \\
black(q_{b/r}, q'_{bx/rx}) \to q'_{bx}, \qquad red(q'_{bx}, q_b) \to q'_{rx}, \\
red(q_{rot2}, q_b) \xrightarrow{|1| = |2|+1} q'_{bx}, \; red(q_b, q'_{bx}) \to q'_{rx}, \\
black(q_{rot1}, q_b) \to q_{rot2}\}
\end{array}
$$

Finally, after `Right-Rotate(T,x->parent->parent)`, we get:

$$
\begin{array}{ll}
A_{15}: F = \{q'_{bx}\}, \Delta = R \cup & \\
\{black(q'_{bx/rx}, q_{b/r}) \to q'_{bx}, & black(q_{b/r}, q'_{bx/rx}) \to q'_{bx} \\
black(q_{b/r}, q_{rot4}) \to q'_{bx}, & black(q_{rot4}, q_{b/r}) \to q'_{bx}, \\
black(q_{rot1}, q_{rot3}) \to q_{rot4}, & red_x(q_b, q_b) \to q_{rot1}, \\
red(q'_{bx}, q_b) \to q'_{rx}, & red(q_b, q'_{bx}) \to q'_{rx}, \\
red(q_{rot4}, q_b) \to q'_{rx}, & red(q_b, q_b) \to q_{rot3}, \\
red(q_b, q_{rot4}) \to q'_{rx}\} &
\end{array}
$$

Then, it can be checked that $\mathscr{L}(A_{15}) \subseteq \mathscr{L}(A_1)$. Case 3 of the insertion procedure is very similar to Case 2 and is omitted.

7 Conclusions

We have presented a method for semi-algorithmic verification of programs that manipulate balanced trees. The approach is based on specifying program pre-conditions, post-conditions, and loop invariants as sets of trees recognised by a novel class of extended tree automata called TASC. TASC come with interesting closure properties and a decidable emptiness problem, and hence are themselves a significant theoretical contribution. Moreover, the semantics of tree-updating programs can be effectively represented as modifications on the internal structures of TASC. The framework has been validated on a case study consisting of the node insertion procedure in a red-black tree. Precisely, we verified that given a balanced red-black tree on the input to the insertion procedure, the output is again a balanced red-black tree.

In the future, we plan to implement the method to be able to perform more case studies. An interesting subject for further research is then extending the method to a fully automatic one. For this, a suitable acceleration method for the reachability computation on TASC is needed. Also, it is interesting to try to generalise the method to handle even the internals of low-level manipulations that temporarily break the tree shape of the considered structures (e.g., by lifting the technique to work over tree automata extended with routing expressions describing additional pointers over the tree backbone).

Acknowledgements We would like to thank Eugene Asarin, Ahmed Bouajjani, Yassine Lakhnech, and Tayssir Touili for their valuable comments.

References

1. R. Alur and P. Madhusudan. Visibly Pushdown Languages. In *Proceedings of STOC'04*. ACM Press, 2004.
2. P. Baldan, A. Corradini, J. Esparza, T. Heindel, B. König, and V. Kozioura. Verifying Red-Black Trees. In *Proceedings of COSMICAH'05*, 2005.
3. M. Barnett, K. Rustan, M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *Proceedings of CASSIS'04*, volume 3362 of *Lectures Notes in Computer Science*. Springer, 2004.
4. A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model-Checking. In *Proceedings of CONCUR'97*, volume 1243 of *Lectures Notes in Computer Science*. Springer, 1997.
5. Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomáš Vojnar. Abstract regular tree model checking of complex dynamic data structures. In *Proceedings of the 13th International Symposium Static Analysis (SAS'06)*, volume 4134 of *Lecture Notes in Computer Science*, pages 52–70, 2006. Springer.
6. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K. Rustan, M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.
7. C. Calcagno, P. Gardner, and U. Zarfaty. Context Logic and Tree Update. In *Proceedings of POPL'05*. ACM Press, 2005.
8. H. Comon-Lundh and V. Cortier. Tree Automata with One Memory, Set Constraints and Cryptographic Protocols. *Theoretical Computer Science*, 331, 2005.
9. H. Comon-Lundh, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications. Available at: *http://www.grappa.univ-lille3.fr/tata*, 1997. Release October 1, 2002.
10. H. Comon-Lundh, F. Jaquemard and N. Perrin. Tree Automata with Memory, Visibility and Structural Constraints. In *Proceedings of FoSSaCS*, volume 4423 of *Lecture Notes in Computer Science*, Springer-Verlag, 2007.
11. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.
12. S. Dal Zilio and D. Lugiez. Multitrees Automata, Presburger's Constraints and Tree Logics. Technical Report 08-2002, LIF, 2002.
13. P.T. Darga and C. Boyapati. Efficient Software Model Checking of Data Structure Properties. In *Proceedings of OOPSLA'06*. ACM Press, 2006.
14. D. Geidmanis. Unsolvability of the Emptiness Problem for Alternating 1-way Multi-head and Multi-tape Finite Automata over Single-letter Alphabet. In *Computers and Artificial Intelligence*, volume 10, 1991.
15. S. Khurshid and D. Marinov. TestEra: Specification-Based Testing of Java Programs Using SAT. *Automated Software Engineering*, 11(4):403–434, 2004.
16. Z. Manna, H. B. Sipma and T. Zhang. Verifying Balanced Trees. In *Proceedings of the Symposium on Logical Foundations of Computer Science (LFCS 2007)*, volume 4514 of *Lecture Notes in Computer Science*, Springer-Verlag, 2007.
17. A. Moeller and M. Schwartzbach. The Pointer Assertion Logic Engine. In *Proceeedings of PLDI'01*. ACM Press, 2001.
18. H. H. Nguyen, C. David, S. Qin and W. N. Chin Automated Verification of Shape and Size Properties via Separation Logic. In *Proceedings of VMCAI'07*, volume 4349 of *Lecture Notes in Computer Science*, Springer-Verlag, 2007.
19. S. Parduhn. Algorithm Animation Using Shape Analysis with Special Regard to Binary Trees. Technical report, Universität des Saarlandes, 2005.
20. H. Petersen. Alternation in Simple Devices. In *Proceedings of ICALP'95*, volume 944 of *Lecture Notes in Computer Science*, Springer-Verlag, 1995.
21. M. Presburger. Über die Vollstandigkeit eines Gewissen Systems der Arithmetik. *Comptes Rendus du I Congrés des Pays Slaves*, Warsaw, 1929.
22. M.O. Rabin. Decidability of Second Order Theories and Automata on Infinite Trees. *Transactions of American Mathematical Society*, 141, 1969.
23. R. Rugina. Quantitative Shape Analysis. In *Proceedings of SAS'04*, volume 3148 of *Lecture Notes in Computer Sciences*. Springer-Verlag, 2004.
24. S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-Valued Logic. *TOPLAS*, 24(3), 2002.

25. H. Seidl, T. Schwentick, A. Muscholl, and P. Habermehl. Counting in Trees for Free. In *Proceedings of ICALP'04*, volume 3142 of *Lecture Notes in Computer Sciences*. Springer-Verlag, 2004.