Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Automata-based Verification of Programs with Tree Updates

Peter Habermehl, Radu Iosif and Tomas Vojnar

Report n^o TR-2005-16

November 3, 2005

Automata-based Verification of Programs with Tree Updates

Peter Habermehl, Radu Iosif and Tomas Vojnar

November 3, 2005

Abstract

This paper describes an effective verification procedure for imperative programs that handle (balanced) tree-like data structures. Since the verification problem considered is undecidable, we appeal to a classical semi-algorithmic approach in which the user has to provide manually the loop invariants in order to check the validity of Hoare triples of the form $\{P\}C\{Q\}$, where P, Q are the sets of states corresponding to the pre- and post-conditions, and C is the program to be verified. We specify the sets of states (representing tree-like memory configurations) using a special class of tree automata named Tree Automata with Size Constraints (TASC). The main advantage of using TASC in program specifications is that they recognize non-regular sets of tree languages such as the *AVL trees*, the *red-black trees*, and in general, specifications involving arithmetic reasoning about the lengths (depths) of various (possibly all) paths in the tree. The class of TASC is closed under the operations of union, intersection and complement, and moreover, the emptiness problem is decidable, which makes it a practical verification tool. We validate our approach considering red-black trees and the insertion procedure, for which we verify that the output of the insertion algorithm is a *balanced* red-black tree, i.e. the longest path is at most twice as long as the shortest path.

Keywords:

Reviewers:

Notes:

How to cite this report:

@techreport { ,
title = { Automata-based Verification of Programs with Tree Updates},
authors = { Peter Habermehl, Radu Iosif and Tomas Vojnar},
institution = { Verimag Technical Report },
number = {TR-2005-16},
year = { },
note = { }
}

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, there arised a number of such search tree structures like the AVL trees, red-black trees, splay trees, and so on [7].

Tree automata [6] are a powerful formalism for specifying sets of trees and reasoning about them. However, one obstacle preventing them from being used currently in program verification is that imperative programs perform destructive updates on selector fields, by temporarily violating the fact that the shape of the dynamic memory is a tree. Another impediment is the fact that tree automata represent regular sets of trees, which is not the case 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 [7] use *tree rotations* (plus the low-level addition/removal of a node to/from a tree) as the only operations that effectively change the structure of the input tree. Such updates are usually implemented as short low-level pointer manipulations [16], which are assumed to be correct in this paper. However, their correctness can be checked separately in a different formalism, such as [17], or by using tree automata extended with additional ”routing” expressions on the tree backbone as in [11].

The second inconvenience has been 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 structure, as for instance the height, or the maximum number of black nodes on all paths, etc. The main advantage of using TASC in program specifications is that they recognize non-regular sets of tree languages, such as the *AVL trees*, the *red-black trees*, and in general, specifications involving arithmetic reasoning about the lengths (depths) of various (possibly all) paths in the tree. We show that the class of TASC is closed under the operations of union, intersection and complement. Also, the emptiness problem is decidable, and the semantics of the programs performing tree updates (node recoloring, rotations, nodes appending/removal) can be effectively represented as changes on the structure of the automata.

Our approach consists in writing pre- and post-condition specifications of a (sequential) imperative program and asking the user to provide loop invariants. The verification problem consists in checking the validity of the invariants and of the Hoare triples of the form $\{P\}C\{Q\}$ where P, Q are the sets of configurations corresponding to the pre- and post-condition, and C is the program to be verified. We validate 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 Verification of programs that handle tree-like structures has attracted researchers with various backgrounds, such as static analysis [12], [16], proof theory [4], and formal language theory [11]. The approach that is the closest to ours is probably the one of PALE (Pointer Assertion Logic Engine) [11], which consists in translating the verification problem into the logic SkS [15] and using tree automata to solve it. Our approach is similar in that we also specify the pre-, post-conditions and the loop invariants, reducing 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. The verification of red-black trees (with balancing) is reported also in [2] by using hyper-graph rewriting systems. Two different approaches, namely net unfoldings, and graph types, are used to check that red nodes have black children and that the tree is balanced, respectively.

The definition of TASC is the result of searching for a class of counter tree automata that combines nice closure properties (union, intersection, complementation) with decidability of the emptiness problem. Existing work on extending tree automata with counters [8], [18] concentrates mostly 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 [13]. This result improves on early work on alternating multi-tape automata recognizing 1-letter languages [9]. However, restricting the number of counters is problematic for obtaining closure of automata under intersection. The solution is to let the actions of the counters depend exclusively on the input tree alphabet, in other words, encode them directly in the input, as size functions. This solution can be seen as a generalization of Visibly Pushdown Languages [1] to trees, for singleton stack alphabets. The general case, with more than one stack symbol, is a subject of future work.

1.1 Running Example

In this section, we introduce our verification methodology for programs using balanced trees. Several data structures based on balanced trees are commonly used, e.g. AVL trees. Here, we will use as a running example red-black trees, which are binary search trees whose nodes are colored by red or black. They are approximately balanced by constraining the way nodes can be colored. The constraints insure that no maximal path can be more than twice as long as any other path. Formally, a node contains an element of an ordered data domain, a color, a left and right pointer and a pointer to its parent. A *red-black tree* is a binary search tree that satifies the following properties:

1. Every node is either red or black.
2. The root is black.
3. Every leaf is black.

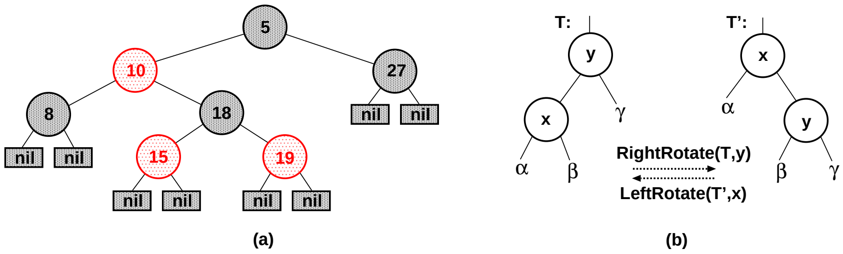

Figure 1: (a) A red-black tree—nodes 10, 15, and 19 are red, (b) the left and right tree rotation

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). Because of the last condition, it is obvious that the set of red-black trees is not regular, i.e. not recognisable by standard tree automata [6]. The main operations on balanced 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 (Figure 1 (b)) which can change the number of black nodes on a given path. The pseudo-code of the inserting operation is the following (see [7]):

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

For this program, we want to show that after an insertion of a node, a red-black tree remains a red-black tree. In this paper, 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 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., recoloring of a node `x->color = red`),
3. left or right rotation (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 node to/from a suitable place that is then followed by the re-balancing operations.

2 Preliminaries

In this paper, 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 the equality sign as syntactic sugar, i.e. $x - y = c \,:\, x - y \leq c \wedge x - y \geq c$. Notice that negation can be eliminated from any formula of $\mathfrak{D}$, since $x - y \not\leq c \iff x - y \geq c + 1$, and so on. 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 $\models \varphi$ meaning that it is valid, i.e. equivalent to true.

The following normal form of formulae from $\mathfrak{D}$ is needed later on, in Section 3.3.

Lemma 1 *Every formula φ of $\mathfrak{D}$ can be effectively written as a disjunction of formulae of the following form, for some suitable indexing $I = \{i_1, i_2, \ldots, i_N\}$ of the free variables:*

$$\bigwedge_{k=1}^{N-1} x_{i_k} - x_{i_{k+1}} \leq c_k \;\wedge \bigwedge_{m \in M \subseteq I} x_m \leq d_m \;\wedge \bigwedge_{p \in P \subseteq I} x_p \geq e_p$$

Proof: We sketch an algorithm that turns any formula of $\mathfrak{D}$ into the normal form. Every time we use the word *choose* in the algorithm below we mean take the disjunction of all possible cases. To obtain the normal form, consider a formula φ written in DNF and choose an indexing $I = \{i_1, i_2, \ldots, i_N\}$ of the variables in φ, conjoining to φ the induced ordering $\theta_I \triangleq x_{i_1} \leq x_{i_2} \leq \ldots \leq x_{i_N}$. For any constraint of the form $x_i - x_j \leq c$ occurring in φ, we apply one of the four cases:

1. if $\theta_I \Rightarrow x_i \leq x_j$ and $c < 0$, there exist $x_i = x_{i_k} \leq x_{i_{k+1}} \leq \ldots \leq x_{i_l} = x_j$ in θ_I. Choose $c \leq c_k, c_{k+1}, \ldots, c_{l-1} \leq 0$ such that $\sum_{s=k}^{l-1} c_s = c$, and replace the constraint $x_i - x_j \leq c$ by the conjunction $\bigwedge_{k \leq s < l} x_{i_s} - x_{i_{s+1}} \leq c_s$ in φ.
2. if $\theta_I \Rightarrow x_i \leq x_j$ and $c \geq 0$, eliminate $x_i - x_j \leq c$ from φ.
3. if $\theta_I \Rightarrow x_i \geq x_j$ and $c < 0$, replace φ by $\perp$.
4. otherwise, when $\theta_I \Rightarrow x_i \geq x_j$ and $c \geq 0$, there exist $x_j = x_{i_k} \leq x_{i_{k+1}} \leq \ldots \leq x_{i_l} = x_i$ in θ_I. Choose $0 \leq c_k, c_{k+1}, \ldots, c_{l-1} \leq c$ such that $\sum_{s=k}^{l-1} c_s = c$, and replace constraint $x_i - x_j \leq c$ by the conjunction $\bigwedge_{k \leq s < l} x_{i_s} - x_{i_{s+1}} \leq c_s$ in φ.

In the resulting formula, replace any conjunction of constraints of the form $x-y \leq c' \wedge x-y \leq c''$ by $x - y \leq \min(c', c'')$. $\square$ The size of the disjunction is exponential in the number of variables, due to the initial choice over all possible orderings, and depends also on the constants c_i, due to the choices of the first and fourth bullet above. Note that this construction doesn't have to be applied if the number of variables is less than or equal to two, which is our case as we show later on. 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. 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 *subtree* of t starting at 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 $P \subseteq \mathbb{N}^*$, we define the *frontier* of P as the set $fr(P) = \{p \in P \mid \forall i \in \mathbb{N}\ pi \notin P\}$. For a tree t, we use $fr(t)$ as a shortcut for $fr(dom(t))$. We denote by $T(\Sigma)$ the set of all trees over the alphabet Σ.

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 hold:*

- $h(\lambda) = \lambda$, *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. $\models \forall x_1 \ldots \forall x_n \ \bigvee_{1 \leq i \leq n} \delta_i \ \wedge \ \bigwedge_{1 \leq i < j \leq n} \neg(\delta_i \wedge \delta_j)$. [1] A *sized alphabet* $(\Sigma, |.|)$ is a ranked alphabet with an associated size function.

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 $f \in \Sigma$, $\#(f) = n$,

[1] 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 $|f(t_1, \ldots, t_n)|$.

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 $a \to q$.

A *run* of A over a tree $t : \mathbb{N}^* \to \Sigma$ is a mapping $\pi : dom(t) \to Q$ 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 $t(p) \to q$.

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

Let us give as example a TASC recognising the set of all balanced red-black trees. Let $\Sigma = \{red, black, nil\}$ with $\#(red) = \#(black) = 2$ and $\#(nil) = 0$. First, we define the size function to be the maximal number of black nodes from the root to a leaf: $|nil| = 1$, $|red(t_1, t_2)| = max(|t_1|, |t_2|)$, and $|black(t_1, t_2)| = max(|t_1|, |t_2|) + 1$. Let $A_{rb} = (\{q_b, q_r\}, \Delta, \{q_b\})$ with $\Delta = \{nil \to q_b, black(q_{b/r}, q_{b/r}) \xrightarrow{|1| = |2|} q_b, red(q_b, q_b) \xrightarrow{|1| = |2|} q_r\}$. 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 $q_{x/y}$.

3 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.

3.1 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 $\mathcal{L}(A) = \mathcal{L}(A_d)$. Concretely, let $A = (Q, \Delta, F)$ and $\mathcal{G}_A$ be the set of all guards labeling the transitions from Δ and $\mathcal{G}_A^n = \{\varphi \in \mathcal{G}_A \mid \|FV(\varphi)\| = n\}$ where $n \in \mathbb{N}$ and $\|FV(\varphi)\|$ denotes the number of free variables in φ. Without loss of generality, we assume that any guard φ labeling a transition of A of the form $f(q_1, \ldots, q_n) \xrightarrow{\varphi} q$ has exactly n free variables.[2] Define $\mathcal{B}_A^n$ as the set of all conjunctions of formulae from $\mathcal{G}_A^n$ and their negations. Let $\mathcal{B}_A = \bigcup_{n \in \mathbb{N}} \mathcal{B}_A^n \cup \{\top\}$. With this notation, define

[2] We can add conjuncts of the form $x_i = x_i$ for all missing variables.

$A_d = (Q_d, \Delta_d, F_d)$ where $Q_d = \mathcal{P}(Q) \times \mathcal{B}_A$, $F_d = \{\langle s, \varphi\rangle \in Q_d \mid s \cap F \neq \emptyset\}$, and:

$$f(\langle s_1,\varphi_1\rangle \ldots \langle s_n,\varphi_n\rangle) \xrightarrow{\varphi} \langle s,\varphi\rangle \in \Delta_d \quad \text{iff} \quad \left\{ \begin{array}{lcl} s & \subseteq & \{q | f(q_1,\ldots,q_n) \xrightarrow{\psi} q \in \Delta, q_i \in s_i\} \text{ and } s \neq \emptyset \\ \varphi & = & \bigwedge \{\psi | f(q_1,\ldots,q_n) \xrightarrow{\psi} q \in \Delta, q_i \in s_i, q \in s\} \wedge \\ & \bigwedge & \{\neg\psi | f(q_1,\ldots,q_n) \xrightarrow{\psi} q \in \Delta, q_i \in s_i, q \in Q \setminus s\} \end{array} \right.$$

$$a \rightarrow \langle s, \top\rangle \in \Delta_d \quad \text{iff} \quad s = \{q \mid a \rightarrow q \in \Delta\}$$

Notice that A_d has no states of the form $\langle s, \bot\rangle$ since they would necessarily be unreachable. The following theorem proves that non-deterministic and deterministic TASC recognize exactly the same languages.

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

Proof: (1) To prove that A_d is deterministic, suppose $t \xrightarrow[A_d]{*} \langle s,\varphi\rangle$ and $t \xrightarrow[A_d]{*} \langle s',\varphi'\rangle$, for some $t \in T(\Sigma)$ and two states $\langle s,\varphi\rangle, \langle s',\varphi'\rangle \in Q_d$. We prove $s = s'$ and $\varphi = \varphi'$ by induction on the structure of t. If $t = a \in \Sigma_0$ we have $s = s' = \{q \in Q \mid a \xrightarrow[A]{} q\}$ and $\varphi = \varphi' = \top$, 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 $\langle s_i,\varphi_i\rangle \in Q_d$ such that $t_i \xrightarrow[A_d]{*} \langle s_i,\varphi_i\rangle$, $1 \leq i \leq n$. 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 viceversa. Let us consider the first case, the other one being symmetric. By the definition of A_d, Δ_d has two rules $f(\langle s_1,\varphi_1\rangle,\ldots,\langle s_n,\varphi_n\rangle) \xrightarrow{\varphi} \langle s,\varphi\rangle$ and $f(\langle s_1,\varphi_1\rangle,\ldots,\langle s_n,\varphi_n\rangle) \xrightarrow{\varphi'} \langle s',\varphi'\rangle$ and A has a rule $f(q_1,\ldots,q_n) \xrightarrow{\psi} q$, for some $q_i \in s_i$, $1 \leq i \leq n$, such that $\varphi \Rightarrow \psi$ and $\varphi' \Rightarrow \neg\psi$. But, since $\langle s,\varphi\rangle$ and $\langle s',\varphi'\rangle$ 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 $s = s'$, and, by definition of A_d, we also have $\varphi = \varphi'$.

(2) "$\mathcal{L}(A_d) \subseteq \mathcal{L}(A)$" We prove that, for all $t \in T(\Sigma)$ and $\langle s,\varphi\rangle \in Q_d$ such that $t \xrightarrow[A_d]{*} \langle s,\varphi\rangle$, 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 $t_i \xrightarrow[A_d]{*} \langle s_i,\varphi_i\rangle$, $1 \leq i \leq n$. By induction hypothesis, for all $q \in s_i$ we have $t_i \xrightarrow[A]{*} q$. By definition of A_d there exists a rule $f(\langle s_1,\varphi_1\rangle,\ldots,\langle s_n,\varphi_n\rangle) \xrightarrow{\varphi} \langle s,\varphi\rangle$, 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 is applicable, 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

$\langle s, \varphi \rangle \in F_d$ then, by the definition of A_d, there exists $q \in s \cap F$. Hence t is accepted by A_d if it is accepted by A. "$\mathcal{L}(A_d) \supseteq \mathcal{L}(A)$" We prove that, for all $t \in T(\Sigma)$ and $q \in Q$, if $t \xrightarrow[A]{*} q$ then there exists $\langle s, \varphi \rangle \in Q_d$ such that $t \xrightarrow[A_d]{*} \langle s, \varphi \rangle$ 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 $q_i \in Q$, $1 \leq i \leq n$. By the induction hypothesis, there exist some $\langle s_i, \varphi_i \rangle \in Q_d$ such that $t_i \xrightarrow[A_d]{*} \langle s_i, \varphi_i \rangle$ 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 $\mathcal{G} = \{\psi \mid \exists q_1 \in s_1, \ldots \exists q_n \in s_n \exists q \in Q\ f(q_1, \ldots, q_n) \xrightarrow[A]{\psi} q\}$. and Γ_ψ be the set of all subsets of $\mathcal{G}$ that contain ψ. By $\Psi_{\mathcal{I}}$ we denote the formula $\bigwedge_{\varphi \in \mathcal{I}} \varphi \wedge \bigwedge_{\varphi \in \mathcal{G} \setminus \mathcal{I}} \neg\varphi$. Obviously $\psi = \bigvee_{\mathcal{I} \in \Gamma_\psi} \Psi_{\mathcal{I}}$. Since $\models \psi(|t_1|, \ldots, |t_n|)$, there exists some $\mathcal{I} \in \Gamma_\psi$ such that $\models \Psi_{\mathcal{I}}(|t_1|, \ldots, |t_n|)$. Now let $s = \{q \mid \exists q_1 \in s_1, \ldots, \exists q_n \in s_n\ f(q_1, \ldots, q_n) \xrightarrow[A]{\psi} q,\ \psi \in \mathcal{I}\}$. and $\varphi = \Psi_{\mathcal{I}}$. Notice that $q \in s$. By the definition of A_d there exists a rule $f(\langle s_1, \varphi_1 \rangle \ldots \langle s_n, \varphi_n \rangle) \xrightarrow{\varphi} \langle s, \varphi \rangle$ in Δ_d, and moreover it is applicable, hence $t \xrightarrow[A_d]{*} \langle s, \varphi \rangle$. By the definition of A_d, if $q \in F$ then $\langle s, \varphi \rangle \in F_d$, hence t is accepted by A if it is accepted by A_d. $\square$

3.2 Union, Intersection and Complementation

In this section, let $A_1 = (Q_1, \Delta_1, F_1)$ and $A_2 = (Q_2, \Delta_2, F_2)$. We can assume w.l.o.g. that Q_1 and Q_2 are disjoint. Then $A_1 \cup A_2 = (Q_1 \cup Q_2, \Delta_1 \cup \Delta_2, F_1 \cup F_2)$. It is easy to check that indeed $\mathcal{L}(A_1 \cup A_2) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$. For intersection, let $A_1 \cap A_2 = (Q_1 \times Q_2, \Delta_{12}, F_1 \times F_2)$, where:

$$f((q_1', q_1''), \ldots, (q_n', q_n'')) \xrightarrow{\varphi' \wedge \varphi''} (q', q'') \in \Delta_{12} \quad \text{iff} \quad f(q_1', \ldots, q_n') \xrightarrow{\varphi'} q' \in \Delta_1 \text{ and}$$
$$f(q_1'', \ldots, q_n'') \xrightarrow{\varphi''} q'' \in \Delta_2$$

The fact that $\mathcal{L}(A_1 \cap A_2) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$ is again an easy check.

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 $\pi \notin Q$ and the following rules, for all $f \in \Sigma$, $q_1, \ldots, q_n \in Q$, where $n = \#(f)$:

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

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

$f(q_1, \ldots, q_n) \xrightarrow[A_c]{\top} q$. It is trivial to check that $\mathcal{L}(A_c) = \mathcal{L}(A)$. Moreover, if A is deterministic, so is A_c.

The *complement* of a deterministic complete TASC $A = (Q, \Delta, F)$ is defined by $\bar{A} = (Q, \Delta, Q \setminus F)$. The proof that $\mathcal{L}(\bar{A}) = T(\Sigma) \setminus \mathcal{L}(A)$ is as in the case of classical tree automata.

3.3 Emptiness

In this section, we give an effective method for deciding emptiness of a TASC. In fact, we address the slightly more general problem: given a TASC $A = (Q, \Delta, F)$ we construct, for each state $q \in Q$, an arithmetic formula $\phi_q(x)$ in one variable that precisely characterizes the sizes of the trees whose roots are labeled with q by A, i.e. $\models \phi_q(n)$ iff $\exists t \; |t| = n$ and $t \xrightarrow[A]{*} q$. As it will turn out, the ϕ_q formulae are expressible in Presburger arithmetic, therefore satisfiability is decidable [14]. This entails the decidability of the emptiness problem, which can be expressed as the satisfiability of the disjunction $\bigvee_{q \in F} \phi_q$.

In order to construct ϕ_q, we shall translate our TASC into an Alternating Pushdown System (APDS) whose stack encodes the value of one integer counter. An APDS is a triple $S = (Q, \Gamma, \delta, F)$ where Q is the set of control locations, Γ is the stack alphabet, F is the set of final control locations, and δ is a mapping from $Q \times \Gamma$ into $\mathcal{P}(\mathcal{P}(Q \times \Gamma^*))$. Notice that APDS do not have an input alphabet since we are interested in the behaviors they generate, rather than the accepted languages. A run of the APDS is a tree $t : \mathbb{N}^* \rightarrow (Q \times \Gamma^*)$ 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 occuring on the frontier are final.

Next, we use the construction of [3] to calculate, for the given set of configurations σ, the set $\mathrm{pre}^*_q(\sigma)$ of configurations with control state q that have a successor set in σ, i.e. $c = \langle q, w \rangle \xrightarrow{*} C \subseteq \sigma$. It is shown in [3] that if σ is a regular language, then so is $\mathrm{pre}^*(\sigma)$, and the alternating finite automaton recognizing the latter can be constructed in time polynomial in the size of the APDS. Hence, the Parikh images of such $\mathrm{pre}^*_q(\sigma)$ sets are semilinear sets definable by Presburger formulae. In our case, $\sigma = \{\langle q, \epsilon \rangle \mid q \in F\}$ is a finite set where ϵ is the (encoding of the) empty stack. Using a unary encoding of the counter (as a stack), we obtain the needed formulae $\phi_q(x)$.

Given a TASC $A = (Q, \Delta, F)$ over an 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 shall represent an integer counter x, by a stack configuration $1^n 0$ if the value of x is $n \in \mathbb{N}$, and $-1^n 0$ if its value is $-n$. The primitive operations on x, i.e. increment, decrement and zero test are encoded by the moves given in Figure 2.

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.

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

Figure 2: Encoding a counter by a stack

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 $f(q_1, \ldots, q_n) \xrightarrow{\varphi} q$, that the current node is of the form $f(t_1, \ldots, t_n)$ with $|f(t_1, \ldots, t_n)| = |t_k| + c_k$, and that $\models \delta_k(|t_1|, \ldots, |t_n|)$, according to the definition of $|.|$. The value $|t_k|$ is said to be the *reference value* of the transition, i.e. the value on which $|f(t_1, \ldots, t_n)|$ actually depends. We shall also consider that $\varphi \wedge \delta_k \in \mathfrak{D}$ has been already converted into the normal form of Lemma 1 that is, a disjunction of formulae of the form $\bigwedge_{s=1}^{n-1} x_{i_s} - x_{i_{s+1}} \leq d_s \wedge \bigwedge_{m \in M} x_m \leq e_m \wedge \bigwedge_{p \in P} x_p \geq l_p$ with $M, P \in \{1, \ldots, n\}$ and $d_s, e_m, l_p \in \mathbb{Z}$.[3]

After each sequence of universal moves, S_A creates n copies of its counter x, let us name them $x_1, \ldots, x_n$. The counter x_i is intended to hold the value $|t_i|$ for $1 \leq i \leq n$, and the counter x holds the value $|f(t_1, \ldots, t_n)|$. Assume that the reference value of the transition is encoded by x_{i_k}, i.e. $x = x_{i_k} + c_{i_k}$. With this notation, Figure 3 (a) shows the alternating moves of S_A that simulate the A-transition considered (for one disjunct of $\varphi \wedge \delta_k$). Figure 3 (b) shows the moves for transitions of the form $a \rightarrow q$.

Filled circles in Figure 3 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. We denote the configurations with control states from $Q \times \Sigma$ by $\langle q, f\rangle^x$ where x is the current value of the counter, and the configurations with control states from Π simply by marking the value of the counter. $\mathrm{sgn}(\ldots)$ denotes the sign function, i.e. $\mathrm{sgn}(n) = 1$ if $n > 0$, $\mathrm{sgn}(0) = 0$ and $\mathrm{sgn}(n) = -1$ if $n < 0$. $\nu_1, \nu_2, \ldots$ are symbolic names for the universal moves performed by S_A.

When simulating the A-transition $f(q_1, \ldots, q_n) \xrightarrow{\varphi} q$, S_A starts with the configuration $\langle q, f\rangle^x$ (Figure 3 (a)). In order to derive the reference value x_{i_k} from x, S_A performs c_{i_k} decrement or increment actions, depending on whether the sign of c_{i_k} is positive or negative. Then S_A performs the universal move ν_1 making three copies of itself. The upper one starts in a state from Π to which two existential (non-deterministic) transitions are attached. The first one decrements the counter an arbitrary number of times in order to obtain some smaller value, while the second moves to a different state starting a sequence of increment/decrement operations of length d_{k-1} in order to obtain the value $x_{i_{k-1}}$ from x_{i_k} (since $x_{i_{k-1}} \leq x_{i_k} + d_{k-1}$). A similar sequence

[3] The case $b_k = 0$ can be treated in a similar way. We only need to guess the reference value, which can be done by a nested increment/decremet loop of the APDS.

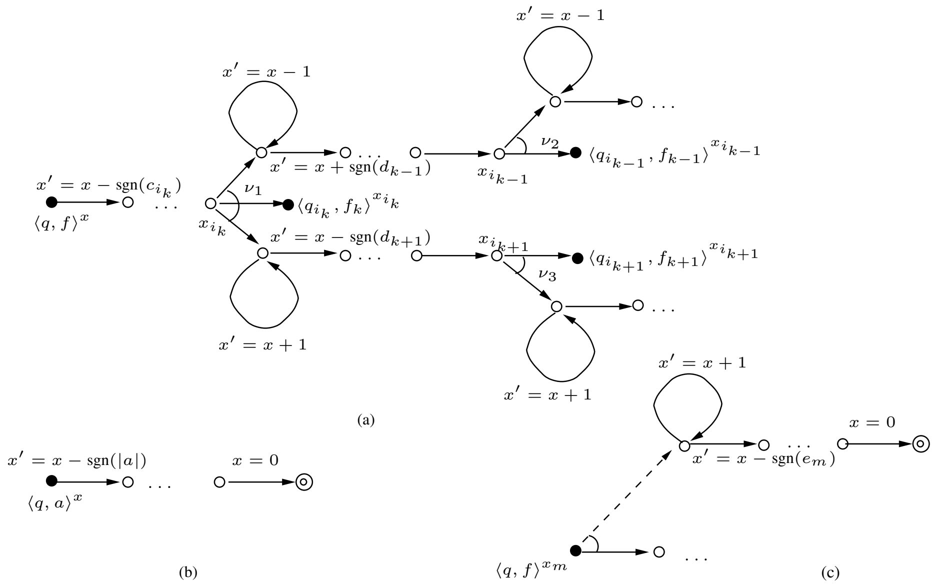

Figure 3: Simulation of a TASC by an APDS

of transitions is performed by the lower universal branch, whereas the middle branch simply changes the control state into $\langle q_{i_k}, f_k \rangle$ without modifying the counter. The symbols f_{k-1}, f_k, f_{k+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_{k-1}, f_k and f_{k+1}, respectivelly. The construction continues until all values $x_{i_1}, x_{i_2}, \ldots, x_{i_n}$ are obtained. Clearly, such values will satisfy the constraint $\varphi \wedge \delta_k$, and all assignments satisfying this formula can be obtained in a run of S_A by iterating the increment/decrement self-loops a sufficient number of times.[4]

In order to simulate moves of the form $a \to q$ (Figure 3 (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 $x = 0$ ensures that S_A accepts only with the empty stack. The universal dotted branch in Figure 3 (c) is used to test that $x_m \leq e_m$ for some $1 \leq m \leq n$. A similar test for $x_p \geq l_p$ can be issued by replacing $x' = x + 1$ with $x' = x - 1$ on the loop. The following lemma is a concretization of the above considerations:

Lemma 2 *For each TASC $A = (Q, \Delta, F)$ over a sized alphabet $(\Sigma, |.|)$ there exists an APDS $S_A = (Q_A, \Gamma, \delta, F_A)$ such that:*

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 $\rho : \mathbb{N}^* \to (Q_A \times \mathbb{N})$ of S_A and a one-to-one tree mapping h between π and ρ such that:*
$$\forall p \in dom(t) \exists q \in Q_A \; \rho(h(p)) = q^{|t_{|p}|} \tag{1}$$

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

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

Moreover, S_A can be effectively constructed from the description of A.

Proof: Let $S_A = (Q_A, \Gamma, \delta, F_A)$ be such that $Q_A = Q \times \Sigma \cup \Pi$, $\Gamma = \{0, -, 1\}$ and $F_A = \{q_f\} \subset \Pi$. For each transition of the form $f(q_1, \ldots, q_n) \xrightarrow{\varphi} q$ from Δ, δ has a distinct set of transitions as in Figure 3 (a) and (c). Transitions of the form $a \rightarrow q$ are translated as in Figure 3 (b).

(1) Let $t \in T(\Sigma)$ be a tree and $\pi : dom(t) \rightarrow Q$ be a run of A on t. We construct ρ and h by the following algorithm. Initially ρ and h are undefined all over $\mathbb{N}^*$, and all positions of $dom(t)$ are unmarked. At each step, ρ' and h' refer to the updated values of ρ and h, respectivelly.

1. **set** $h'(\lambda) = \lambda$ and $\rho'(\lambda) = \langle \pi(\lambda), t(\lambda) \rangle^{|t|}$.
2. **repeat**
   choose some unmarked $p \in \{p \in dom(t) \mid \#(t(p)) = n > 0 \text{ and } \rho(h(p)) = \langle \pi(p), t(p) \rangle^{|t_{|p}|}\}$
   (a) consider a run θ of S_A, starting in state $\rho(h(p))$ such that $fr(\theta) = \{s_1, \ldots, s_n\}$ and $\theta(s_i) = \langle \pi(pi), t(pi) \rangle^{|t_{pi}|}$, for all $1 \le i \le n$.
   (b) extend h and ρ as follows: for all $1 \le i \le n$, $h'(pi) = h(p) \cdot s_i$, and for all $s \in dom(\theta)$, $\rho'(h(p) \cdot s) = \theta(s)$.
   (c) mark p as visited.

   until no more positions match.

Note that it is always possible to find θ at step (2.a), due to the construction in Figure 3 (a). Suppose that the reference value of the transition is k, $1 \le k \le n$, i.e. $|t_{|p}| = |t_{|pk}| + c_k$ for some $c_k \in \mathbb{Z}$. Moreover, since the A-transition $f(q_1, \ldots, q_n) \xrightarrow{\varphi} q$ is possible for $t = f(t_1, \ldots, t_n)$ and $t_i = f_i(\ldots)$, $1 \le i \le n$, it is the case that $\models \varphi(|t_1|, \ldots, |t_n|)$. Also, due to the definition of $|.|$, we have $\models \delta_k(|t_1|, \ldots, |t_n|)$. The normal form of Lemma 1 of $\varphi \wedge \delta_k$ ensures that $\models \bigwedge_{s=1}^{n-1} |t_{i_s}| - |t_{i_{s+1}}| \le d_i \wedge \bigwedge |t_m| \le e_m \wedge \bigwedge |t_p| \ge l_p$ with $d_i, e_m, l_p \in \mathbb{Z}$. By sufficiently iterating the increment/decrement self-loops on the universal branches of S_A one can obtain the values $|t_i|$ from $|t|$, which guarantees that $\theta(s_i) = \langle \pi(pi), t(pi) \rangle^{|t_{pi}|}$, for all $1 \le i \le n$.

The algorithm terminates, since $dom(t)$ is finite. Moreover, h is a tree mapping, according to Definition 1. It is one-to-one since at no point in the algorithm two different locations from $dom(\pi)$ are mapped into the same location from $dom(\rho)$. By induction on the length of p we can prove that: for all $p \in dom(t)$, $\rho(h(p)) = \langle \pi(p), t(p) \rangle^{|t_{|p}|}$, which trivially implies (1). Since, for all $p \in fr(t)$ we have that $\rho(h(p)) = \langle q, a \rangle^{|a|}$, with $a \rightarrow q \in \Delta$, for those positions, $\rho(h(p))$ leads uniquely to the accepting configuration q_f^0, according to Figure 3 (b). The other positions on $fr(\rho)$ correspond to branches testing $x_m \diamond e_m$ in Figure 3 (c), which also lead to q_f^0. This proves that ρ is accepting.

(2) Let $\rho : \mathbb{N}^* \rightarrow (Q_A \times \mathbb{N})$ be an accepting run of S_A. We construct t, π and h using the following algorithm. Initially t, π and h are undefined all over $\mathbb{N}^*$, and all positions from $dom(\rho)$ are unmarked. At each step, t', π' and h' refer to the updated values of t, π and h, respectivelly.

1. **for** $\rho(\lambda) = \langle q, f \rangle^u$ **set** $t'(\lambda) = f$, $\pi'(\lambda) = q$ and $h'(\lambda) = \lambda$.

2. **repeat**
choose some unmarked $p \in \{p \in dom(\rho) \mid \rho(p) = \langle q, f\rangle^u\}$

   (a) let $S = \{s \in dom(\rho_{|p}) \mid \rho_{|p}(s) \in Q \times \Sigma \times \mathbb{N}$ and $\forall s' \prec s\ \rho_{|p}(s') \in \Pi \times \mathbb{N}\}$. For each $s \in S$ let v_s denote the difference between the number of increment and the number of decrement operations performed along the branch leading from p to ps. Fix some indexing of $S = \{s_1, \ldots, s_n\}$. Choose an A-transition $f(q_1, \ldots, q_n) \xrightarrow{\varphi} q \in \Delta$ and k a number between 1 and n be such that $|f(t_1, \ldots, t_n)| = |t_k| + c_k$, with side condition δ_k. Let $\mathcal{I} = \{i_1, \ldots, i_n\}$ be an indexing of the free variables $\{x_1, \ldots, x_n\}$ such that $v_{s_k} = d_{i_k}$ and $\models (\varphi \wedge \delta_k)[x_{i_1}/v_{s_1}, \ldots, x_{i_n}/v_{s_n}]$.

   (b) for all $s_i \in S$ such that $\rho(ps) = \langle q_i, f_i\rangle^{u_i}$, $1 \leq i \leq n$ **set** $t'(pi) = f_i$, $\pi'(pi) = q_i$ and $h'(pi) = ps_i$.

   (c) mark p as visited.

   until no more positions match.

Note that it is always possible to find the set S of positions corresponding to the start points of a (bottom-up) A-transition, since every sequence of S_A-moves as in Figure 3 (a) corresponds to some A-transition. The rest is finding the matching A-transition, which can be done by enumerating them, since Δ is finite. The algorithm terminates, because $dom(\rho)$ is finte.

By induction on the height of $t_{|p}$, we prove that, for all $p \in dom(t)$, $\rho(h(p)) = \langle \pi(p), t(p)\rangle^{|t_{|p}|}$. For the base case $p \in fr(t)$ (the height of $t_{|p}$ is one) we have that $\rho(h(p)) = \langle q, a\rangle^{|a|}$ for some transition $a \rightarrow q \in \Delta$, by the construction in Figure 3 (b) and the fact that ρ is accepting. For the induction step, suppose $\rho(pi) = \langle q_i, f_i\rangle^{|t_{|pi}|}$ for all $1 \leq i \leq n$, $f(q_1, \ldots, q_n) \xrightarrow{\varphi} q \in \Delta$ and $|f(t_1, \ldots, t_n)| = |t_k| + c_k$, $\models \delta_k(|t_1|, \ldots, |t_n|)$ for some $1 \leq k \leq n$. By the construction in Figure 3 (a), we have that $\rho(h(p)) = \langle q, f\rangle^{|t_k|+c_k}$. This guarantees that condition (1) is satisfied.

Moreover, $\models \varphi(|t_1|, \ldots, |t_n|)$ is the case, so the A-transition is enabled. By the same inductive argument, this guarantees that π is a valid run of A. It is easy to see that, h is a tree mapping and it is one-to-one. $\square$

As a remark, the decidability of the emptiness problem for TASC can be also proved via a reduction to the class of *tree automata with one memory* [5] 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.

4 Semantics of Tree Updates

As explained in Section 1.1, 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 a pointer variable is pointing to in the tree, (2) testing or changing certain data fields of the encountered tree nodes, such as the color 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 2 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 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 minimization, 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).

4.1 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 only of the form $f(q_1, q_2) \xrightarrow{|1| - |2| = b} q$ with $b \in \mathbb{Z}$.

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 rTASC is again an rTASC, and that applying the determinisation of Section 3.1 to an rTASC yields another rTASC. Moreover, the intersection of an rTASC with a classical tree automaton is again an rTASC.[5] Clearly, rTASC are not closed under complementation, as inequality guards are not allowed.

Minimization of rTASC. The simple form of the guards allows us to have a practical minimization procedure based on the minimization for classical bottom-up tree automata [6]. 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 by the following conditions:

- $dom(t) = dom(\delta(t))$,
- 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 $\delta(t)(p) = \langle t(p), |t_{|p1}| - |t_{|p2}|\rangle$.

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

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

- $f(q_1, q_2) \xrightarrow{|1| - |2| = b} q$ 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 3 *Given an rTASC A over a sized alphabet $(\Sigma, |.|)$, for all trees $t \in T(\Sigma)$, we have $t \in \mathcal{L}(A)$ if and only if $\delta(t) \in \mathcal{L}(A_\delta)$.*

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 $t_i \xrightarrow[A]{*} q_i$, $1 \le i \le 2$. 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. $\Box$

Now given an rTASC A we compute A_δ, minimize it using the classical construction from [6], obtaining ${A_\delta}^{min}$. The minimal rTASC A^{min} is obtained by performing the reverse operation on ${A_\delta}^{min}$, i.e. 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' recognizing the same language, i.e. $\mathcal{L}(A) = \mathcal{L}(A^{min}) = \mathcal{L}(A')$. Then, $\delta(\mathcal{L}(A)) = \delta(\mathcal{L}(A')) = \mathcal{L}(A'_\delta) = \mathcal{L}({A_\delta}^{min})$. Since A' and A'_δ have the same number of states, we contradict the minimality of ${A_\delta}^{min}$.

4.2 Representing Sets of Memory Configurations

Let us consider a finite set of *pointer variables* $\mathcal{V} = \{x, y, \ldots\}$ and a disjoint finite set of data values $\mathcal{D}$, e.g. $\mathcal{D} = \{red, black\}$. In the following, we let $\Sigma = \mathcal{P}(\mathcal{V} \cup \mathcal{D} \cup \{nil\})$ where nil indicates a null pointer value. The arity function is defined as follows: $\#(f) = 2$ if $nil \notin f$, and $\#(f) = 0$ otherwise. For a tree $t \in T(\Sigma)$ and a variable $x \in \mathcal{V}$, we say that a position $p \in dom(t)$ is *pointed to by* x if and only if $x \in t(p)$.

For the rest, 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 $p \in dom(t)$ such that $x \in t(p)$. This condition can be ensured by the construction of A: let $Q = \mathcal{Q} \times \mathcal{P}(\mathcal{V})$ and Δ consist only of rules of the form $f(\langle q_1, v_1\rangle, \langle q_2, v_2\rangle) \xrightarrow{\varphi} \langle q, v\rangle$ where (1) $v = (f \cup v_1 \cup v_2) \cap \mathcal{V}$ and (2) $f \cap v_1 = f \cap v_2 = v_1 \cap v_2 = \emptyset$. Intuitively, a control state $\langle q, v\rangle$ "remembers" all variables encountered by condition (1), while condition (2) ensures that no variable is encountered twice.

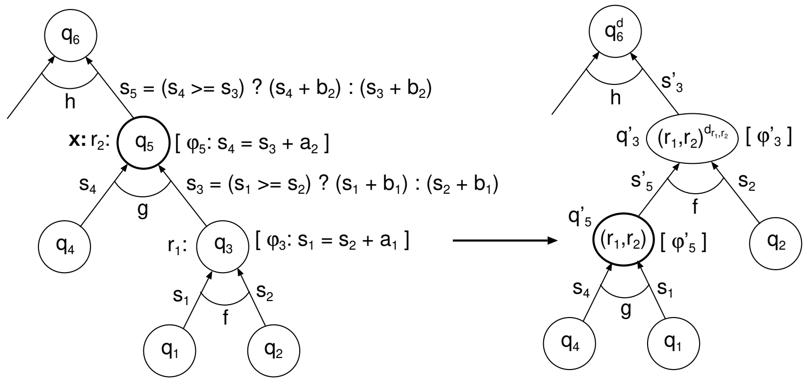

Figure 4: Left rotation on an rTASC

4.3 Modeling Tree Rotations

Let $x \in \mathcal{V}$ be a fixed variable. We shall construct an rTASC $A' = (Q', \Delta', F')$ that describes the set of trees that are the result of the *left rotation* of a tree from $L(A)$ applied at the node pointed to by x. The case of the right tree rotation is very similar.[6] In the description, we will be referring to Figure 4 illustrating the problem.

Let $R_x = \{(r_1, r_2) \in \Delta^2 \mid x \in g \ \wedge \ r_1 : f(q_1, q_2) \xrightarrow{\varphi_3} q_3 \ \wedge \ r_2 : g(q_4, q_3) \xrightarrow{\varphi_5} q_5\}$ be the set of all the pairs of automata rules that can yield a rotation, and be modified because of it. Other rules may then have to be modified to reflect the change in one of their *left hand side states*, e.g. the change of q_5 to q_3' in the h-rule in Figure 4, or to reflect the *change in the balance* that may result from the rotation, i.e. a change in the *difference of the sizes* of the subtrees of some node. We discuss later what changes in the balance can appear after a rotation, and Lemma 4 proves that the set D of the possible changes in the balance in the described trees is finite. The automaton A' can thus be constructed from A as follows:

1. $Q' = Q \cup R_x \cup (R_x \times D) \cup (Q \times D)$ where we add new states for the rotated parts and to reflect the changes in the balance.

2. $\Delta' = \Delta \cup \Delta_r \cup \beta(\Delta \cup \Delta_a)$ where:

    - Δ_r corresponds to the rotated rules is the smallest set such that for all $(r_1, r_2) \in R_x$ where $r_1 : f(q_1, q_2) \xrightarrow{\varphi_3} q_3$ and $r_2 : g(q_4, q_3) \xrightarrow{\varphi_5} q_5$, Δ_r contains the rules $g(q_4, q_1) \xrightarrow{\varphi_5'} q_5'$ and $f(q_5', q_2) \xrightarrow{\varphi_3'} q_3'$ where $q_5' = (r_1, r_2)$ and $q_3' = (r_1, r_2)^{d_{r_1,r_2}}$. Here, we use $(r_1, r_2)^{d_{r_1,r_2}}$ as a shorthand for $\langle (r_1, r_2), d_{r_1,r_2} \rangle$. The value $d_{r_1,r_2} \in \mathbb{Z}$ represents the change in the balance caused by the rotation based on r_1, r_2. We describe the computation of φ_3', φ_5', and d_{r_1,r_2} below.

[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.

- Δ_a is the set of rules that could be applied just above the position where a rotation takes place. For each $(r_1, r_2) \in R_x$, we take all rules from Δ that have q_5 within the left hand side and add them to Δ_a, with (r_1, r_2) substituted for q_5.
- β (described in detail in Section 4.4) is the function that implements the necessary changes in the guards and input/output states (adding the d-component) of the rules due to the changes in the balance.

3. $F' = (F \times D) \cup F_r$. Here, F_r captures the case where q'_3 becomes accepting, i.e. the right child of the node previously labeled by q_3 becomes the root of the entire tree.

Suppose that φ_3 is $|t_1| = |t_2| + a_1$ and let us denote the sizes of the sub-trees read at q_1 and q_2 before the rotation by s_1 and s_2, respectivelly. 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 labeled by q_3 before the rotation. Also, suppose that φ_5 is $|t_1| = |t_2| + a_2$ 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 labeled 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 φ'_3, φ'_5, and d_{r_1,r_2} 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_{r_1,r_2} 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 φ'_5 relating a subtree of size s_4 and s_1 (cf. Figure 4) is $|t_1| = |t_2| + b_1 + a_2$.
2. For $a_1 < 0$, we have $s_4 = s_1 - a_1 + b_1 + a_2$, and so φ'_5 is $|t_1| = |t_2| - a_1 + b_1 + a_2$.

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 $\varphi'_3 : |t_1| = |t_2| + a_1 + b_1 + a_2 + b_2$.
    (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 $\varphi'_3 : |t_1| = |t_2| + b_1 + a_2 + b_2$.

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 $\varphi'_3 : |t_1| = |t_2| + a_1 + b_2$.

The computation of the change in the balance d_{r_1,r_2} is similar to the above. The first case to be considered is $\Phi_{4\geq3} : s_4 \geq s_3 \iff a_2 \geq 0$. 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:

$$s'_3 = \begin{cases} \text{if } \Phi_{4\geq1} : \\ \quad \begin{cases} \text{if } s_4 + b_2 \geq s_2 : s_4 + b_2 + b_1 \\ \text{if } s_4 + b_2 < s_2 : s_2 + b_1 \end{cases} \\ \text{if } \Phi_{4<1} : \\ \quad \begin{cases} \text{if } s_1 + b_2 \geq s_2 : s_1 + b_2 + b_1 \\ \text{if } s_1 + b_2 < s2 : s_2 + b_1 \end{cases} \end{cases}$$

Let us first consider the subcase when $\Phi_{4\geq1}$. 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 $d_{r_1,r_2} = b_1$.

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 $d_{r_1,r_2} = -a_1 - a_2 - b_2$.

   (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 $d_{r_1,r_2} = -a_2 - b_2$.

The remaining cases of the d_{r_1,r_2} computation are similar to the above.

4.4 Propagating Changes in the Balance through rTASC

As said, tree updates such as recoloring or rotations may introduce changes in the balance at certain points. These changes may affect the balance at all positions above the considered node. The role of the β function is to propagate a change in balance d upwards in the trees recognized by the rTASC. The way β changes a set of rules is illustrated in Figure 5. For every $d \in D$, every input rule $f(q_1, q_2) \xrightarrow{\varphi} q_3$ is changed to two rules $f(q_1^d, q_2) \xrightarrow{\varphi'} q_3^{d'}$ and $f(q_1, q_2^d) \xrightarrow{\varphi''} q_3^{d''}$ corresponding to the cases when the change in the 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 new guards are $\varphi' : |t_1| = |t_2| + a + d$ and

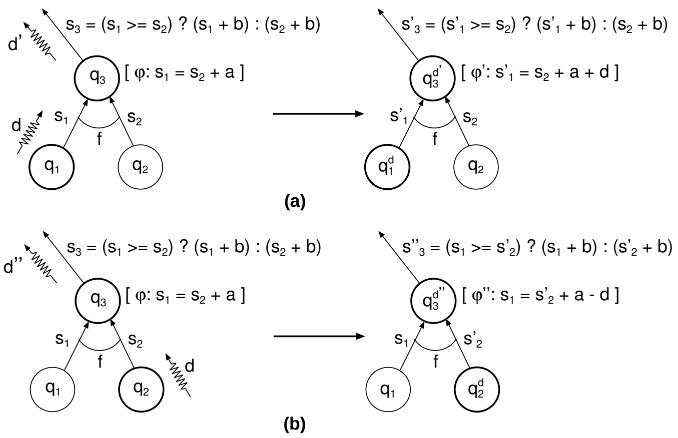

Figure 5: Propagation of changes in the balance in an rTASC

$\varphi'' : |t_1| = |t_2| + a - d$. Let us further analyse the changes in the balance propagated upwards after d comes from the bottom.

Suppose the change in balance is coming from the left as in Figure 5 (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 $d' = 0$.

Similarly, when the change is coming from the right, as in Figure 5 (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 $d'' = d$.

When a change d in the size happens at a child node, at its parent, the change is either eliminated, d' or d'' is 0, stays the same, d' or d'' equals d, becomes $-|a|$ (note that $a \geq 0$ for $d' = -a$, and $a < 0$, for $d'' = a$), or finally, becomes $-|a|+d$. We can now close our construction by showing that the set D of possible changes in the sizes is finite.

Lemma 4 *For an rTASC A over a set of variables $\mathcal{V}$ and a variable $x \in \mathcal{V}$, the set D of the possible changes in the balance generated by a left tree rotation at x is finite.*

Proof: For D to be infinite, there would have to be a possibility to start with some initial change (either some $-|a|$ or some d_{r_1,r_2}), and then keep modifying it infinitely many times. This can happen only when we use infinitely many times the last case (i.e. $-|a| + d$) from the previous paragraph. Then, we can only start with some d_{r_1,r_2} as for this case to be applied, we need the change in the size at a child node to be positive ($a \geq 0 \ \wedge \ a - d < 0$ for the right case, and $a < 0 \wedge a+d \geq 0$ for the left case). Note that every time the considered case of propagating the change in the size is applied, we have $d' < d$ or $d'' \leq d$ meaning that the change in the size either does not change or decreases. However, this means that we cannot get an unbounded number of different changes because sooner or later we reach zero and stop generating further changes.□

Note that when we allow the use of two different constants b_f^1 and b_f^2 in the size function for binary nodes, the resulting class of automata will not be closed with respect to left or right rotations. It may happen that the changes in the balance could diverge, thus we would need an infinite number of compensating constants to be used for the different heights of the possible trees.

4.5 Other Operations on Sets of Trees Described by rTASC

Let us now briefly show that in addition to the tree rotations, rTASC are closed with respect to all other operations that we commonly need when dealing with balanced binary trees too. We have listed these operations in Section 1.1. Due to space constraints, we are only giving an informal description of these operations here.

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->left == x`. In general, we consider any test of the form e_1==e_2, where $e_{1,2}$ are of the form v->n_1->n_2->...n_m with $v \in \mathcal{V}$, $m \in \mathbb{N}$, and $n_1, ..., n_m \in \{\texttt{left}, \texttt{right}, \texttt{parent}\}$. Suppose we are given an rTASC A recognizing 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.

A formal description of this construction can be found in the full version of the paper. Here, let us present an example of A_c for the condition `x->parent->left == x` that should clearly illustrate the construction. We will have rules $f \to q_1$ and $g \to q_2$ for every $f, g \in \Sigma$ such that $\texttt{x} \in g \setminus f$. We recall that $\Sigma = \mathcal{P}(\mathcal{V} \cup \mathcal{D})$. Then, we have rules $f(q_1, q_1) \to q_1$, $g(q_1, q_1) \to q_2$, $f(q_2, q_1) \to q_3$, $f(q_3, q_1) \to q_3$, and $f(q_1, q_3) \to q_3$ for q_3 being the only accepting state. Here, the pointer referencing pattern gets simply captured in the rule $f(q_2, q_1) \to q_3$.

Second, pointer assignments of the form v' = v->n_1-> n_2->...n_m can be handled by our method, using a simple transformation of the input rTASC which 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 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 pinter expression of the form v->n_1->n_2->...n_m is an analogy of the pointer reference checking and pointer assignments. However, by changing the data contents of some node (e.g., we recolour some node in a red-black tree), we can change the size of the appropriate subtree. In this case, we have to use the function β from Section 4.4 to reflect the change in the balance in the guards of all the rules that can be fired above the node that changed.

Next, when thinking of the physical insertion of a new leaf node, note 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. Inserting a new leaf memory node 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. To encode the operation, we modify the input rTASC by first non-deterministically marking some null node with a pointer variable, i.e. we change its label from $\{\mathtt{null}\}$ to $\{\mathtt{null}, x\}$. Then, we replace all rules $\{\mathtt{null}, x\} \to q_x$ by rules $\{\mathtt{null}\} \to q_{\mathtt{null}}$, $\{d, x\}(q_{\mathtt{null}}, q_{\mathtt{null}}) \xrightarrow{|1| = |2|} q_x$ where d models the initial data content. The addition of the new symbol may change the size of the subtrees above q_x (as, e.g., adding a black node in a red-black tree), and so we have to use the function β from Section 4.4 to adjust the guards of the influenced rules.

Finally, the deletion of a frontier node pointed to by some pointer variable y is modeled by removing the rules $\{d, y\}(q, q_{\mathtt{null}}) \xrightarrow{\varphi} q_y$. (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 function β from Section 4.4 to handle the changes in the balance resulting from a deletion of a node.

5 Case Study: Red-Black Tree Insertion

To illustrate our methodology, we show how to prove an invariant for the main loop in procedure RB-Insert. (Note that all the steps are normally to be done fully automatically.) This invariant is needed to prove the correctness of the insertion procedure given in Section 1.1 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 1.1.

For presentation purposes, if no guard is specified on a binary rule, we assume it to be $|1| = |2|$. Also, we denote singleton sets by their unique element, e.g. $\{red\}$ by red, and d_x stands for

$\{d, x\}$, where $d \in \{red, black, nil\}$. Let $R = \{nil \to q_b, red(q_b, q_b) \to q_r, black(q_{b/r}, q_{b/r}) \to q_b\}$. The loop invariant is given by the following rTASC A_1.

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

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 1.1.

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

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

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 $\mathcal{L}(A_8) \subseteq \mathcal{L}(A_1)$.

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:

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

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 $red(q_b, q_b) \rightarrow q_{rx}$ and rule (12) to $red_x(q_b, q_{rx}) \rightarrow q''_{rx}$.

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.

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

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

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

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

$$A_{14}: \ F = \{q'_{bx}\}, \Delta = R \ \cup \{$$
$$black(q'_{bx/rx}, q_{b/r}) \rightarrow q'_{bx}, \quad red_x(q_b, q_b) \rightarrow q_{rot1},$$
$$black(q_{b/r}, q'_{bx/rx}) \rightarrow q'_{bx}, \quad red(q'_{bx}, q_b) \rightarrow q'_{rx},$$
$$red(q_{rot2}, q_b) \xrightarrow{|1| = |2| + 1} q'_{bx}, \ red(q_b, q'_{bx}) \rightarrow q'_{rx}, \quad black(q_{rot1}, q_b) \rightarrow q_{rot2}\}$$

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

$$A_{15}: \ F = \{q'_{bx}\}, \Delta = R \ \cup \{ \ black(q'_{bx/rx}, q_{b/r}) \rightarrow q'_{bx}, \quad black(q_{b/r}, q'_{bx/rx}) \rightarrow q'_{bx}$$
$$black(q_{b/r}, q_{rot4}) \rightarrow q'_{bx}, \quad black(q_{rot4}, q_{b/r}) \rightarrow q'_{bx}, \quad black(q_{rot1}, q_{rot3}) \rightarrow q_{rot4},$$
$$red_x(q_b, q_b) \rightarrow q_{rot1}, \quad red(q'_{bx}, q_b) \rightarrow q'_{rx}, \quad red(q_b, q'_{bx}) \rightarrow q'_{rx},$$
$$red(q_{rot4}, q_b) \rightarrow q'_{rx}, \quad red(q_b, q_b) \rightarrow q_{rot3}, \quad red(q_b, q_{rot4}) \rightarrow q'_{rx}\}$$

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

6 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 invariants as sets of trees recognized by a novel class of extended tree automata called TASC. TASC come with interesting closure properties and a decidable emptiness problem. 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 verify 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 generalize 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).

Acknowledgment. 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. 1

[2] P. Baldan, A. Corradini, J. Esparza, T. Heindel, B. König, and V. Kozioura. Verifying Red-Black Trees. In *Proc. of COSMICAH'05*, 2005. 1

[3] A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model-Checking. In *Proceedings of CONCUR '97*, volume 1243 of *LNCS*. Springer, 1997. 3.3

[4] C. Calcagno, P. Gardner, and U. Zarfaty. Context Logic and Tree Update. In *Proceedings of POPL'05*. ACM Press, 2005. 1

[5] H. Comon and V. Cortier. Tree Automata with One Memory, Set Constraints and Cryptographic Protocols. *Theoretical Computer Science*, 331, 2005. 3.3

[6] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 1997. Release October 1, 2002. 1, 1.1, 4.1, 4.1

[7] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990. 1, 1.1

[8] S. Dal Zilio and D. Lugiez. Multitrees Automata, Presburger's Constraints and Tree Logics. Technical Report 08-2002, LIF, 2002. 1

[9] 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. 1

[10] P. Habermehl, R. Iosif, and T. Vojnar. Automata-based Verification of Programs with Tree Updates. Technical Report TR-2005-16, Verimag, 2005.

[11] A. Moeller and M. Schwartzbach. The Pointer Assertion Logic Engine. In *Proceeedings of PLDI'01*. ACM Press, 2001. 1, 1

[12] S. Parduhn. Algorithm Animation Using Shape Analysis with Special Regard to Binary Trees. Technical report, Universität des Saarlandes, 2005. 1

[13] H. Petersen. Alternation in Simple Devices. In *Proceedings of ICALP'95*, volume 944 of *LNCS*. Springer, 1995. 1

[14] M. Presburger. Über die Vollstandigkeit eines Gewissen Systems der Arithmetik. *Comptes Rendus du I Congrés des Pays Slaves*, Warsaw, 1929. 3.3

[15] M.O. Rabin. Decidability of Second Order Theories and Automata on Infinite Trees. *Transactions of American Mathematical Society*, 141, 1969. 1

[16] R. Rugina. Quantitative Shape Analysis. In *Proceedings of SAS'04*, volume 3148 of *LNCS*. Springer, 2004. 1, 1

[17] S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-Valued Logic. *TOPLAS*, 24(3), 2002. 1

[18] H. Seidl, T. Schwentick, A. Muscholl, and P. Habermehl. Counting in Trees for Free. In *Proceedings of ICALP'04*, volume 3142 of *LNCS*. Springer, 2004.

1