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

Proving Termination of Tree Manipulating Programs

Peter Habermehl, Radu Iosif, Adam Rogalewicz, Tomáš Vojnar

Report n^{o} TR-2007-1

February 14, 2007

Proving Termination of Tree Manipulating Programs

Peter Habermehl, Radu Iosif, Adam Rogalewicz, Tomáš Vojnar

February 14, 2007

Abstract

We consider the termination problem of programs manipulating tree-like dynamic data structures. Our approach is based on an abstract-check-refine loop. We use abstract regular tree model-checking to infer invariants of the program. Then, we translate the program to a counter automaton which simulates it. If the counter automaton can be shown to terminate using existing techniques, the program terminates. If not, we analyze the possible counterexample given by a counter automata termination checker and either conclude that the program does not terminate, or else refine the abstraction and repeat. We show that the spuriousness problem for lasso-shaped counterexamples is decidable in some non-trivial cases. We applied the method successfully on several interesting case studies.

Keywords:

Reviewers:

Notes:

How to cite this report:

@techreport { ,
title = { Proving Termination of Tree Manipulating Programs},
authors = { Peter Habermehl, Radu Iosif, Adam Rogalewicz, Tomáš Vojnar},
institution = { Verimag Technical Report },
number = {TR-2007-1},
year = { },
note = { }
}

1 Introduction

Verification of programs with dynamic linked data structures is a difficult task, among other reasons, due to the use of unbounded memory, and the intricate nature of pointer manipulations. Most of the approaches existing in this area concentrate on checking safety properties such as, e.g., absence of null pointer dereferences, preservation of shape invariants, etc. In this paper, we go further and tackle the universal termination problem of programs manipulating tree data structures. Namely, we are interested in proving that such a program terminates for any input tree out of a given set described as an infinite regular tree language over a finite alphabet.

We handle sequential, non-recursive programs working on trees with parent pointers and data values from a finite domain. The basic statements we consider are data assignments, non-destructive pointer assignments, and tree rotations. This is sufficient for verifying termination of many practical programs over tree-shaped data structures (e.g., AVL trees or red-black trees) used, in general, for storage and a fast retrieval of data. Moreover, many programs working on singly- and doubly-linked lists fit into our framework as well. We do not consider dynamic allocation in this version of the paper, but insertion/removal of leaf nodes, common in many practical tree manipulating programs, can be easily added, if not used in a loop.

We build on *Abstract Regular Tree Model Checking* (ARTMC), a generic framework for symbolic verification of infinite-state systems, based on representing regular sets of configurations by finite tree automata, and program statements as operations on tree automata. We represent a given program as a control flow graph whose nodes are annotated with (overapproximations of) sets of reachable configurations computed using ARTMC. From the annotated control flow graph, we build a counter automaton (CA) that simulates the program. The counters of the CA keep track of different measures within the working tree: the distances from the root to nodes pointed to by certain variables, the sizes of the subtrees below such nodes, and the numbers of nodes with a certain data value. Termination of the CA is analyzed by existing tools, e.g., [13, 17, 31].

Our analysis uses an *Abstract-Check-Refine* loop. If the tool we use to prove termination of the CA succeeds, this implies that the program terminates on any input from the given set. Otherwise, the CA checker tool outputs a lasso-shaped counterexample. For the class of CA generated by our translation scheme, we prove that it is decidable whether there exists a non-terminating run of the CA over the given lasso[1].

However, even if we are given a real lasso in the generated CA, due to the abstraction involved in its construction, we still do not know whether this implies also non-termination of the program. We then map the lasso over the generated CA back into a lasso in the control of the program, and distinguish two cases. If (1) the program lasso does not contain tree rotations, termination of all computations along this path is decidable. Otherwise, (2) if the lasso contains tree rotations, we can decide termination under the additional assumption that there exists a CA (not necessarily known to us) that witnesses termination of the program (i.e., intuitively, in the case when the tree measures we use are strong enough to show termination). In both cases, if the program lasso

[1]If the analyzer used returns a spurious lasso-shaped counterexample for the termination of the CA, we suggest choosing another tool.

is found to be spurious, we refine the abstraction and generate a new CA from which an entire family of counterexamples (including this particular one) is excluded.

The analysis loop is not guaranteed to terminate even if the given program terminates due to the fact that our problem is, in fact, not recursively enumerable. However, experience with our implementation of a prototype tool shows that the method is successfully applicable to proving termination of various real-life programs.

Related Work. The area of research on automated verification of programs manipulating dynamic linked data structures is recently quite live. Various approaches to verification of such programs differing in their principles, degree of automation, generality, and scalability have been proposed. The approaches are based on, e.g. monadic second order logic [29], 3-valued predicate logic with transitive closure [32], separation logic [30, 21], other logics [11, 3, 27], automata [7, 5, 19, 9], or other symbolic representations such as [35, 4, 25, 14]. With few exceptions, all existing methods tackle verification of safety properties. In [34], specialized ranking functions over the number of nodes reachable from pointer variables were used to verify termination of programs manipulating linked lists. Termination of programs manipulating lists has further been considered in [20, 6] using constraints on the number of nodes in particular list segments not having internal nodes pointed from outside. In [6], which is the closest in spirit to the work presented here, it is shown that such programs without data stored in the lists may be modeled by bisimilar counter automata with locations based on abstract shapes of the lists reachable by the programs. Termination of programs manipulating trees has been considered in [26] where the Deutsch-Schorr-Waite tree traversal algorithm was proved to terminate using a manually created progress monitor, encoded in first order logic.

Plan of the Report. In Section 2, we introduce some basic notions about trees, tree automata, relations, and transducers. In Section 3, we describe the syntax and semantics of the programs we work with (including a description of a simple pre-processing phase that we apply over the input programs for some technical reasons related to our technique). In Section 4, we introduce our abstract-check-refine loop. We briefly explain some basics of abstract regular tree model checking in Section 5. Next, in Section 6, we describe the way we translate the considered programs into counter automata. In Section 7, we discuss how spuriousness of lasso-shaped counterexamples can be checked over the generated counter automata. Subsequently, in Section 8, we explain how an abstraction that yields a spurious lasso-shaped counterexample may be refined. In Section 9, we discuss a prototype implementation of our technique and the experimental results that we obtained from it. Finally, we conclude in Section 10.

2 Preliminaries

Trees. Given a finite set of *colors* $\mathcal{C}$ we define the *binary alphabet* $\Sigma_{\mathcal{C}} = \mathcal{C} \cup \{\square\}$ where the *arity* function is $\#(c) = 2$ and $\#(\square) = 0$. Π denotes the set of tree positions $\{0,1\}^*$, i.e., the set of all sequences of 0's and 1's. Let $\lambda \in \Pi$ denote the empty sequence, and $p.q$ denote the concatenation of sequences $p,q \in \Pi$. $p \preceq q$ denotes the fact that p is a prefix of q. A *binary tree* t over $\mathcal{C}$ is a partial mapping $t : \Pi \to \Sigma_{\mathcal{C}}$ such that $dom(t)$ is a finite prefix-closed subset of Π, and for each $p \in dom(t)$:

- if $\#(t(p)) = 0$, then $t(p.0) = t(p.1) = \perp$,
- otherwise, if $\#(t(p)) = 2$, then $p.0, p.1 \in dom(t)$.

When writing $t(p) = \perp$, we mean that t is undefined at position p. Let t_ε be the empty tree, $t_\varepsilon(p) = \perp$ for all $p \in \Pi$. 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. $t[p \leftarrow c]$ denotes the tree that is labelled as t, except at position p where it is labelled with c. $t\{p \leftarrow u\}$ denotes the tree obtained from t by replacing the $t_{|p}$ subtree with u. We denote by $\mathcal{T}(\mathcal{C})$ the set of all trees over the binary alphabet $\Sigma_{\mathcal{C}}$.

Tree Automata. A *tree automaton* [15] over a binary alphabet $\Sigma_{\mathcal{C}}$ is a tuple $A = (Q, \Sigma_{\mathcal{C}}, F, \Delta)$ where Q is a set of states, $F \subseteq Q$ is a set of final states, and Δ is a set of transition rules of the form:

- $\square \rightarrow q$ or
- $c(q_1, q_2) \rightarrow q$, $c \in \mathcal{C}$.

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

- if $\#(t(p)) = 0$ (i.e., if $t(p) = \square$), then $\square \rightarrow q \in \Delta$,
- otherwise, if $\#(t(p)) = 2$ and $q_i = \pi(p.i)$ for $i \in \{0, 1\}$, then $t(p)(q_0, q_1) \rightarrow q \in \Delta$.

A run π is said to be *accepting* if and only if $\pi(\lambda) \in F$. The *language* of A, denoted as $\mathcal{L}(A)$, is the set of all trees over which A has an accepting run. Let $A = (Q, \Sigma_{\mathcal{C}}, F, \Delta)$ be a tree automaton and $q \in Q$ a state, then we define the *language of the state* q—$\mathcal{L}(A, q)$—as the set of trees $\mathcal{L}(A_q)$ where $A_q = (Q, \Sigma_{\mathcal{C}}, \{q\}, \Delta)$.

Regularity-Preserving Tree Relations. A relation $\rho \subseteq \mathcal{T}(\mathcal{C}) \times \mathcal{T}(\mathcal{C})$ is a *regularity-preserving tree relation* iff for each regular tree language $\mathcal{L}(A)$ described by some tree automaton A over the alphabet $\Sigma_{\mathcal{C}}$, there exists a tree automaton A' over $\Sigma_{\mathcal{C}}$ such that $\mathcal{L}(A') = \rho(\mathcal{L}(A)) = \{t' \in \mathcal{T}(\mathcal{C}) \mid \langle t, t' \rangle \in \rho\}$. For a sequence of regularity-preserving tree relations $\Omega = \rho_1, \rho_2, \ldots, \rho_n$ where $\rho_i \subseteq \mathcal{T}(\mathcal{C}) \times \mathcal{T}(\mathcal{C})$ for $1 \leq i \leq n$, $n \geq 1$, and a tree automaton A over the alphabet $\Sigma_{\mathcal{C}}$, we define $\Omega(\mathcal{L}(A)) = \rho_n(\ldots(\rho_2(\rho_1(\mathcal{L}(A)))))$. We also denote by $\rho^*(\mathcal{L}(A))$ the reflexive and transitive closure of the language $\mathcal{L}(A)$ according to the relation ρ.

Tree Transducers. A structure-preserving *tree transducer* [15] over a binary alphabet $\Sigma_{\mathcal{C}}$ is a tuple $\tau = (Q, \Sigma_{\mathcal{C}}, F, \Delta)$ where Q is a set of states, $F \subseteq Q$ is a set of final states, and Δ is a set of transition rules of the form:

- $\square \rightarrow \langle \square, q \rangle$ or
- $c_1(q_1, q_2) \rightarrow \langle c_2, q \rangle$, $c_1, c_2 \in \mathcal{C}$.

A *run* of τ over a tree $t : \Pi \to \Sigma_C$ is a mapping $\pi : dom(t) \to \Sigma_C \times Q$ such that for each position $p \in dom(t)$ where $\langle c, q\rangle = \pi(p)$, we have:

- if $\#(t(p)) = 0$ (i.e., if $t(p) = \square$), then $\square \to \langle \square, q\rangle \in \Delta$,
- otherwise, if $\#(t(p)) = 2$ and $\langle c_i, q_i\rangle = \pi(p.i)$ for $i \in \{0,1\}$, then $t(p)(q_0, q_1) \to \langle c, q\rangle \in \Delta$.

A run π is said to be *accepting* iff $\varsigma(\pi(\lambda)) \in F$ where $\varsigma(\langle c, q\rangle) = q$ for $q \in Q$ and $c \in \Sigma_C$. For a given tree $t \in \mathcal{T}(\mathcal{C})$, we define the set of trees $\tau(t) \subseteq \mathcal{T}(\mathcal{C})$, resulting from the transduction by τ, as $\tau(t) = \{t' \in \mathcal{T}(\mathcal{C}) \mid \exists \pi : \varsigma(\pi(\lambda)) \in F \wedge t'(p) = \phi(\pi(t(p)))\}$ where $\phi(\langle c, q\rangle) = c$ for $q \in Q$ and $c \in \Sigma_C$. This is, $\tau(t)$ is a (possibly empty) set of trees given by all possible accepting runs of the transducer τ on the tree t. A transducer τ over an alphabet Σ_C represents the relation $\rho(\tau) \subseteq \mathcal{T}(\mathcal{C}) \times \mathcal{T}(\mathcal{C})$ defined as $\rho(\tau) = \{(t, t') \in \mathcal{T}(\mathcal{C}) \times \mathcal{T}(\mathcal{C}) \mid t' \in \tau(t)\}$. The relations represented by structure-preserving tree transducers are a special case of regularity-preserving relations. For a tree automaton A over an alphabet Σ_C, we denote by $\tau(\mathcal{L}(A))$ the effectively computable regular tree language $\rho(\tau)(\mathcal{L}(A))$, which may be represented by some tree automaton A', $\mathcal{L}(A') = \rho(\tau)(\mathcal{L}(A))$.

Counter Automata. For an arithmetic formula φ, let $FV(\varphi)$ denote the set of free variables of φ. For a set of variables X, let $\Phi(X)$ denote the set of arithmetic formulae with free variables from $X \cup X'$ where $X' = \{x' \mid x \in X\}$. If $\nu : X \to \mathbb{Z}$ is an assignment of $FV(\varphi) \subseteq X$, we denote by $\nu \models \varphi$ the fact that ν is a satisfying assignment of φ.

A counter automaton (CA) is a tuple $A = \langle X, Q, q_0, \varphi_0, \to\rangle$ where X is the set of counters, Q is a finite set of control locations, $q_0 \in Q$ is a designated initial location, φ_0 is an arithmetic formula such that $FV(\varphi_0) \subseteq X$, describing the initial assignments of the counters, and $\to \in Q \times \Phi(X) \times Q$ is the set of transition rules.

A configuration of a CA is a pair $\langle q, \nu\rangle \in Q \times (X \to \mathbb{Z})$. The set of all configurations is denoted by $\mathfrak{C}$. The transition relation $\xrightarrow[A]{\varphi} \subseteq \mathfrak{C} \times \mathfrak{C}$ is defined by $(q, \nu) \xrightarrow[A]{\varphi} (q', \nu')$ iff there exists a transition $q \xrightarrow{\varphi} q'$ such that if σ is an assignment of $FV(\varphi)$ where $\sigma(x) = \nu(x)$ and $\sigma(x') = \nu'(x)$, we have that $\sigma \models \varphi$ and $\nu(x) = \nu'(x)$ for all variables x with $x' \notin FV(\varphi)$. We denote by $\xrightarrow[A]{}$ the union $\bigcup_{\varphi \in \Phi} \xrightarrow[A]{\varphi}$, and by $\xrightarrow[A]{*}$ the reflexive and transitive closure of $\xrightarrow[A]{}$. A *run* of A is a sequence of configurations $(q_0, \nu_0), (q_1, \nu_1), (q_2, \nu_2) \ldots$ such that $(q_i, \nu_i) \xrightarrow[A]{} (q_{i+1}, \nu_{i+1})$ for each $i \geq 0$ and $\nu_0 \models \varphi_0$. We denote by $\mathfrak{R}_A$ the set of all configurations reachable by A, i.e., $\mathfrak{R}_A = \{(q, \nu) \mid (q_0, \nu_0) \xrightarrow[A]{*} (q, \nu)$ for some $\nu_0 \models \varphi_0\}$.

3 Programs with Trees

Program Syntax. We consider sequential, non-recursive C-like programs whose abstract syntax is given in Figure 1(a). Here, *Lab* is a finite set of program labels (control locations), *PVar* a finite set of pointer variables, and *Data* is a finite data domain. We assume that the programs work on tree-shaped data structures whose nodes contain a data element `data` with a value from the set *Data* and three next pointers, namely `left`, `right`, and `up`. For technical reasons, we require w.l.o.g. that there is no statements taking the control back to the initial line.

$l \in Lab,\ x,y,z \in PVar,\ d,e,f \in Data$
$Program := \{l : Stmnt;\}^*$
$Stmnt := IfStmnt \mid Update \mid Asgn \mid Rotate$
$IfStmnt := \texttt{if}\ (Cond)\ \texttt{then goto}\ l_1;$
$\qquad \texttt{else goto}\ l_2;$
$Cond := x == y \mid x == \texttt{null} \mid$
$\qquad x.data == d$
$Field := \texttt{left} \mid \texttt{right} \mid \texttt{up}$
$Asgn := x = y\{.Field\} \mid x = \texttt{null} \mid$
$\qquad x.\texttt{data} = d$
$Rotate := \texttt{leftRotate}(x) \mid \texttt{rightRotate}(x)$

```
x = root;
while (x != null) do
  if (x.left != null && x.left.data != marked)
  then x = x.left;
  else
    if (x.right != null && x.right.data != marked)
    then x = x.right;
    else x.data = marked;
         x = x.up;
```

Figure 1: (a) Abstract syntax of programs with trees and (b) a depth-first tree traversal procedure

Programs manipulating singly-linked lists, doubly-linked lists, and trees without parent pointers may be considered a special case of the programs considered here if we suffice with the statements from Figure 1(a). We add syntactic sugar by allowing nesting of pointer expressions as, e.g., in the expression `x.left.data`, nesting of statements in conditional statements, composed conditions in conditional statements, and the use of `while` statements. We consider the depth-first traversal (DFT) of a tree shown in Figure 1(b) as the running example of this paper.

Concrete Semantics. Let $\mathcal{C} = Data \times 2^{PVar}$ be a set of colors and $\Sigma_{\mathcal{C}} = \mathcal{C} \cup \{\square\}$ a binary alphabet. For any symbol $\sigma = \langle d, S\rangle \in \mathcal{C}$, we denote $\delta(\sigma) = d$ and $\nu(\sigma) = S$. By convention, $\delta(\square) = \perp$ and $\nu(\square) = \emptyset$. A concrete program state is a pair $s = \langle l,t\rangle$ where $l \in Lab$ is the current program label, and $t : \Pi \to \Sigma$ is the current working tree. We require that each variable from *PVar* belongs to the set $\nu(t(p))$ for at most one position $p \in dom(t)$. Intuitively, memory configurations of the considered programs are represented as trees where a node is either null and represented by $\square$, or it contains a data value and a set of pointer variables pointing to it.

We consider a special error state denoted by *Err*. In the state $\langle l,t\rangle$, a condition $x == \texttt{null}$ in a conditional statement evaluates to true (and leads to a change of the control to the appropriately labelled line) iff $x \notin \nu(t(p))$ for all $p \in dom(t)$. A condition $\mathtt{x} == \mathtt{y}$ evaluates to true iff $x \in \nu(t(p)) \Leftrightarrow y \in \nu(t(p))$ for all $p \in dom(t)$. If $x \in \nu(t(p))$ for some $p \in dom(t)$, a condition $\mathtt{x}.data == d$ evaluates to true if $\delta(t(p)) = d$ and to false if $\delta(t(p)) \neq d$, else if $x \notin \nu(t(p))$ for all $p \in dom(t)$, its evaluation takes the control into *Err*. The concrete semantics of the most interesting assignment and rotation statements from Figure 1 is described by the rules shown in Figure 3 where $u_{left} = t_\varepsilon\{\lambda \leftarrow t_{|p.1}\}\{0 \leftarrow t_{|p}\}\{0.1 \leftarrow t_{|p.1.0}\}$, and $u_{right} = t_\varepsilon\{\lambda \leftarrow t_{|p.0}\}\{1 \leftarrow t_{|p}\}\{1.0 \leftarrow t_{|p.0.1}\}$. The semantics of the remaining assignment and rotation statements is straightforward.

Given a program P, we denote by $\langle l,t\rangle \xrightarrow[P]{s} \langle l',t'\rangle$ the fact that P has a transition leading from the first state to the second one caused by firing a statement s. We denote by $\xrightarrow[P]{}$ the union

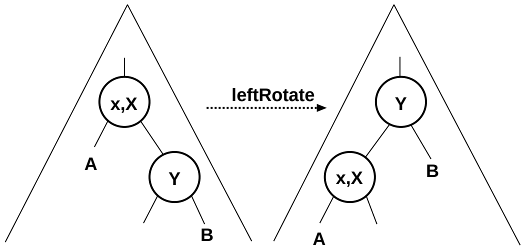

Figure 2: The left tree rotation

$$\frac{\forall p \in dom(t) : x \notin \nu(t(p))}{\langle l,t\rangle \to \langle l',t\rangle} \qquad \frac{p \in dom(t) \quad x \in \nu(t(p))}{\langle l,t\rangle \to \langle l', t[p \leftarrow \langle \delta(t(p)), \nu(t(p)) \setminus \{x\}\rangle]\rangle}\; x = \texttt{null}$$

$$\frac{\begin{array}{c} p \in dom(t) \quad y \in \nu(t(p)) \\ t(p.0) \neq \Box \quad \langle l,t\rangle \xrightarrow{x=null} \langle l',t'\rangle \end{array}}{\langle l,t\rangle \to \langle l', t'[p.0 \leftarrow \langle \delta(t(p.0)), \nu(t(p.0)) \cup \{x\}\rangle]\rangle} \qquad \frac{\begin{array}{c} p \in dom(t) \quad y \in \nu(t(p)) \\ t(p.0) = \Box \quad \langle l,t\rangle \xrightarrow{x=null} \langle l',t'\rangle \end{array}}{\langle l,t\rangle \to \langle l',t'\rangle} \qquad \frac{\forall p \in dom(t) \,.\, y \notin \nu(t(p))}{\langle l,t\rangle \to Err}\; x = y.\texttt{left}$$

$$\frac{p \in dom(t) \quad x \in \nu(t(p))}{\langle l,t\rangle \to \langle l', t[p \leftarrow \langle d, \nu(t(p))\rangle]\rangle} \qquad \frac{\forall p \in dom(t) \,.\, x \notin \nu(t(p))}{\langle l,t\rangle \to Err}\; x.\texttt{data} = d$$

$$\frac{p \in dom(t) \quad x \in \nu(t(p)) \quad t(p.1) \neq \Box}{\langle l,t\rangle \to \langle l', t\{p \leftarrow u_{left}\}\rangle} \qquad \frac{\begin{array}{c} (\forall p \in dom(t) \,.\, x \notin \nu(t(p))) \vee \\ (p \in dom(t) \wedge x \in \nu(t(p)) \wedge t(p.1) = \Box) \end{array}}{\langle l,t\rangle \to Err}\; \texttt{leftRotate}(x)$$

Figure 3: The concrete semantics of program statements

$\bigcup_{s \in Stmnt} \xrightarrow[P]{s}$, and by $\xrightarrow[P]{*}$ the reflexive and transitive closure of $\xrightarrow[P]{}$. For a program statement $s \in Stmnt$, let $post(s,I) = \{t' \mid \exists t \in I \,.\, t \xrightarrow[P]{s} t'\}$.

3.1 Program Preprocessing

Some of the considered program statements may influence the counters of the CA that we build to simulate programs in several different ways. For instance, after $x = x.\texttt{left}$, the distance of x from the root may increase by one, but it may also become undefined (which we represent by the special value -1) if $x.\texttt{left}$ is null. Similarly, a single rotation statement may change the distance of a node pointed by some variable from the root in several different ways according to where the node is located in a concrete input tree. In particular, if some variable points into a subtree that moves up in a rotation, its distance from the root will decrease, whereas if the same variable does not point into this subtree, its distance from the root will not be modified (or it can increase if it points into the subtree moving down due to the rotation)—cf. Figure 2.

For technical reasons related to our abstraction refinement scheme, we need a one-to-one mapping between actions of a program and the counter manipulations simulating them. In order to ensure the existence of such a mapping, we decompose each program statement s into a finite set of *instructions* $I(s)$ with disjoint guards such that (1) the union of the semantics of these instructions is the semantics of the original statement, i.e., $post(s,T) = \bigcup_{i \in I(s)} post(i,T)$ for any set of trees T, and (2) each instruction i is represented in the counter automaton by a unique counter operation denoted by $\theta(i)$.

In particular, statements $x = \texttt{null}$ and $x = y$ correspond directly to instructions. Conditional statements of the form $x == \texttt{null}$ and $x == y$ are decomposed into two instructions each corresponding to their true and false branches. A conditional statement $x.\texttt{data} == d$ is decomposed into three instructions corresponding to its true and false branches and an error branch for the case $x == \texttt{null}$.

Every statement $x = y.\texttt{left}$ is split to three instructions, namely $\texttt{goLeftNull}(x,y)$ assigning null to x iff y is not null and $y.\texttt{left}$ is null, $\texttt{goLeftNonNull}(x,y)$ setting x to $y.\texttt{left}$ iff both y and $y.\texttt{left}$ are not null, and $\texttt{goLeftErr}(x,y)$ iff y is null. This mainly distinguishes the cases when the distance of a variable from the root is increasing from when it becomes null. Similarly, we split each statement $x = y.\texttt{right}$ into instructions $\texttt{goRight}\{\texttt{Null}|\texttt{NonNull}|\texttt{Err}\}(x,y)$ and each statement $x = y.\texttt{up}$ into instructions $\texttt{goUp}\{\texttt{Null}|\texttt{NonNull}|\texttt{Err}\}(x,y)$.

Further, we split each statement $x.\texttt{data} = d_1$ into $\texttt{changeDataErr}(x)$ for the case of x being null and $|Data|$ instructions $\texttt{changeData}(x,d_1,d_2)$ changing the value of $x.\texttt{data}$ from d_1 to d_2 for each $d_2 \in Data$. This explicitly shows the number of which data elements in the working tree is decreasing, resp. increasing.

For each choice of disjoint subsets $X,Y,A,B \subseteq PVar$ with $x \in X$, we split each statement $\texttt{leftRotate}(x)$ into instructions $\texttt{leftRotate}(x,X,Y,A,B)$ (see Figure 2). The set X contains variables that are aliased to x, Y contains variables pointing to the right son of x, A contains variables pointing inside the left subtree of x, and B contains variables pointing into the right subtree of the right son of x. Intuitively, counters corresponding to variables from the same set (and from $Pvar \setminus (X \cup Y \cup A \cup B)$) will be updated in the same way in the counter automaton (cf. Figure 2). In particular,

- for variables from X, the root distance will increase by one, and the subtree size will decrease by at least one,
- for variables from Y, the root distance will decrease by one, and the subtree size will increase by at least one,
- for variables from A, the root distance will increase by one, and the subtree size will stay the same,
- for variables from B, the root distance will decrease by one, and the subtree size will stay the same,
- for all other variables from $Pvar \setminus (X \cup Y \cup A \cup B)$, which may point either into the left subtree of the right son of x, point into nodes not below nor equal to x, or be null, the root distance as well as the subtree size do not change.

Formally, the semantics of $\texttt{leftRotate}(x,X,Y,A,B)$ is the same as the one of $\texttt{leftRotate}(x)$ with the following conditions added into the preconditions: $X = \nu(t(p))$, $Y = \nu(t(p.1))$, $A = \cup_{p.0.p' \in dom(t)} \nu(t(p.0.p'))$, and $B = \cup_{p.1.1.p' \in dom(t)} \nu(t(p.1.1.p'))$ where $x \in \nu(t(p))$. We then also add an instruction `leftRotateErr`(x) for the case of x or its right son being null. The `rightRotate`(x) statements are split analogously.

In general, the number of instructions corresponding to a single statement may be exponentially large in the number of program variables. However, in practice, the use of program invariants (see Section 6) will greatly restrict the number of instructions in the model. Let $Instr = \bigcup_{s \in Stmnt} I(s)$ be the set of all instructions. We introduce the notation $\xrightarrow[P]{i}$ for $i \in Instr$ in a way analogous to $\xrightarrow[P]{s}$ for $s \in Stmnt$.

4 The Termination Analysis Loop

Our termination analysis procedure based on abstraction refinement is schematically depicted in Figure 4. We start with the control flow graph (CFG) of the given program and use ARTMC (described briefly in Section 5) to generate invariants for the control points of the CFG. If the user wants, ARTMC may also be used to check many further safety properties of the programs (shape invariants, data-oriented properties, etc.) [7, 9]. Subsequently, the CFG annotated with the invariants (an abstract CFG—ACFG, see Section 6) is converted into a CA[2], which is checked for termination using an existing tool (e.g., [31]). If the CA is proved to terminate, termination of the program is proved too. Otherwise, the termination analyzer outputs a lasso-shaped counterexample. We check whether this counterexample is real in the CA—if not, we suggest the use of another CA termination checker (for brevity, we skip this in Figure 4). If the counterexample is real on the CA, it is translated back into a sequence of program instructions and analyzed for spuriousness on the program. If the counterexample is found to be real even there, the procedure reports non-termination. Otherwise, the program CFG is refined by splitting some of its nodes (actually, the sets of program configurations associated with certain control locations), and the loop is reiterated. Moreover, ARTMC may also be re-run to refine the invariants used (as briefly discussed in Section 8).

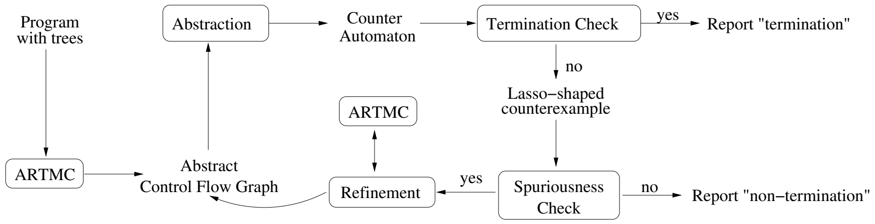

Figure 4: The proposed abstract-check-refine loop

The pseudocode in Figure 5 presents the abstract-check-refine loop that we propose in a more detailed view. The fact that the inner-most loop of our procedure terminates for some K (pro-

[2]The use of invariants in the ACFGs allows us to remove impossible transitions and therefore improves the accuracy of the translation to CA.

vided no destructive updates are used in the loop, or there exists some counter automaton of the form considered by us that may be used to show termination of the given program) is proved in Section 7. The number of times ACFGs may be refined purely by splitting their control locations is given by the (user-defined) constant $MaxSteps$. If $MaxSteps$ of refining ACFGs by splitting their locations are not enough, we re-run ARTMC with a more precise abstraction in order to refine the program line invariants from which ACFGs are built. In such a case, ARTMC is re-run on the refined CFG underlying the last computed ACFG in order to preserve the effect of the splitting steps done so-far.

```
AbstractionHeight := 1;  // The ARTMC precisness parameter (for the finite-height abstraction schema).
F := the control flow graph of P;
while (true) {
        G := ARTMC(AbstractionHeight, F, Init);
          // Use ARTMC to compute the ACFG G over the CFG F for the initial set of trees Init.
        if a safety error was found, report it and exit;
        N := 1;
        while (N ≤ MaxSteps) {
                A := translateACFGintoCA(G);
                test termination of A using an existing CA termination checker TC;
                if TC reports termination for A, report termination of P and exit;
                let S.L be the lasso-shaped counterexample for A;  // S – stem, L – loop
                if A terminates on S.L, but TC fails to show this, report that TC is not strong enough and exit;
                S_P.L_P := translateLassoFromCABackToProgram(S.L);
                K := 1; I := the set of trees reachable via S_P from Init;
                while (true) {
                        test non-termination for the trees of height K from Init;
                        if non-termination is witnessed, report non-termination and exit;
                        I := the set of trees reachable via L_P from I;
                        if (I == ∅) break;  // The lasso breaks for all input trees after at most K iterations.
                        K := K + 1;
                }
                G := refineACFG(G, S_P.L_P, K);
                N := N + 1;
        }
        AbstractionHeight := AbstractionHeight + 1;
        F := the refined CFG (some nodes are split) behind the current ACFG G;
}
```

Figure 5: A detailed view of the proposed abstract-check-refine loop

If our termination analysis stops with either a positive or a negative answer, the answer is exact. However, we do not guarantee termination for any of these cases. Indeed, this is the best we can achieve as the problem we handle is not recursively enumerable even when destructive updates (i.e., tree rotations) are not allowed. This can be proved by a reduction from the complement of the halting problem for 2-counter automata [28].

Theorem 1 *The problem whether a program with trees without destructive updates terminates on any input tree is not recursively enumerable.*

Proof: A 2-counter machine M with non-negative counters c_1, c_2 is a sequential program:

$$0: \texttt{ins}_1; 1: \texttt{ins}_2; \cdots; n-1: \texttt{ins}_n;$$

where $\texttt{ins}_n$ is a `halt` instruction and $\texttt{ins}_i$ with $i = 1,2,\cdots,n-1$ are instructions of the following two types, for $0 \le k, k_1, k_2 < n$, and $1 \le j \le 2$:

1. $c_j = c_j + 1; \texttt{goto}\ k;$
2. $\texttt{if}\ c_j = 0$ then goto k_1 else $\big(c_j = c_j - 1;$ goto $k_2\big);$

We use the fact that the complement of the halting problem for 2-counter machines is not recursively enumerable, and reduce this problem to the universal halting problem for programs with trees. Let M be a 2-counter machine. We built a program P working with a singly-linked list[3] l of length n that simulates M and terminates iff M, starting with both counters at 0, can perform at least n steps. P will have five variables $I_{1,2}$, $D_{1,2}$ and E, initially pointing to the head of the list. The control structure of P is almost the same as the one of M, and the statements of P are as follows:

1. $c'_j = c_j + 1; \texttt{goto}\ k$; where $j = 1,2$ is simulated by:

   I_j = I_j.`next`;
   E = E.`next`;
   if (E = `null`) goto STOP; else goto k;

2. $\texttt{if}\ c_j = 0$ then goto k_1 else $\big(c_j = c_j - 1;$ goto $k_2\big)$; where $j = 1,2$ is simulated by:

   if ($I_j = D_j$) E = E.`next`; else goto l;
   if (E = `null`) goto STOP; else goto k_1;
   l: D_i = D_i.`next`;
   E = E.`next`;
   if (E = `null`) goto STOP; else goto k_2;

3. halt is simulated by:

   goto LOOP;

where STOP is a halting location, and LOOP is a location with a self-loop in P. P first tests if M can perform the next action, and in case the test fails, it goes into a self loop. Otherwise, when M has performed a number of actions equal to the length of the list, it stops. Then P stops on any input list iff M runs forever. But the latter problem is the complement of the halting problem, known not to be r.e. This concludes our proof. □

Therefore we do not further discuss termination guarantees for our analysis procedure in this paper, and postpone the research on potential partial guarantees, in some restricted cases, for the future. However, despite the theoretical limits, the experimental results reported in Section 9 indicate a practical usefulness of our approach.

[3]In fact trees are not even needed for the proof.

5 Abstract Regular Tree Model Checking

In this section, we first briefly recall the basic principles of abstract regular tree model checking (ARTMC) [8]—the original description is a bit modified to meet our specific needs. Then we propose how ARTMC can, in particular, be applied within our termination checking procedure.

5.1 Basics of ARTMC

In *regular tree model checking* [33, 10, 1, 2, 8], configurations of the systems being examined are encoded as trees over a suitable ranked alphabet, sets of configurations are described by tree automata, and transitions of the systems are encoded as tree transducers (or, sometimes, using some special-purpose operations over tree automata [9]). Subsequently, one computes the set of all configurations reachable from an initial set of configurations by repeatedly applying the tree transducers (or the operations replacing them) on the set of the so-far reached configurations (encoded as tree automata). In order to make the computation converge as often as possible, various *acceleration methods* are used. One of the most successful acceleration techniques used in regular tree model checking is abstraction that is the basis of the *abstract regular tree model checking* (ARTMC) [8].

More formally, let $\mathcal{C}$ be a set of colors and $\mathbb{M}_\Sigma$ the set of all tree automata over the binary alphabet $\Sigma_\mathcal{C}$. Let $Init \in \mathbb{M}_\Sigma$ be a tree automaton describing the set of initial configurations of the system being examined, and let $Bad \in \mathbb{M}_\Sigma$ be a tree automaton describing a set of bad configurations of the system. Further, let the behaviour of the examined system be described by a regularity-preserving tree relation $\rho \subseteq \mathcal{T}(\mathcal{C}) \times \mathcal{T}(\mathcal{C})$ such that the image of any regular set $\mathcal{L} \subseteq \mathcal{T}(\mathcal{C})$ by ρ is effectively computable and, moreover, such that the image of any regular set $\mathcal{L}' \subseteq \mathcal{T}(\mathcal{C})$ by ρ^{-1} is computable as well. (Relations described by structure-preserving tree transducers, which we will use to express most of the pointer manipulations—the only exception being tree rotations, are a special case of such relations.) Finally, suppose that the verification problem is to check whether:

$$\rho^*(\mathcal{L}(Init)) \cap \mathcal{L}(Bad) = \emptyset \tag{1}$$

In abstract regular tree model checking, the above problem is solved by using a CEGAR loop to compute a safe overapproximation of the set of reachable configurations $\rho^*(\mathcal{L}(Init))$.

Abstraction in ARTMC is applied over tree automata encoding sets of reachable configurations of the examined systems. We define an abstraction function as a mapping $\alpha : \mathbb{M}_\Sigma \to \mathbb{A}_\Sigma$ where $\mathbb{A}_\Sigma \subseteq \mathbb{M}_\Sigma$ and $\forall M \in \mathbb{M}_\Sigma : \mathcal{L}(M) \subseteq \mathcal{L}(\alpha(M))$. An abstraction α' is called a *refinement* of the abstraction α if $\forall M \in \mathbb{M}_\Sigma : \mathcal{L}(\alpha'(M)) \subseteq \mathcal{L}(\alpha(M))$. Given a regular tree relation ρ and an abstraction α, we define a mapping $\rho_\alpha : \mathbb{M}_\Sigma \to \mathbb{M}_\Sigma$ as $\forall M \in \mathbb{M}_\Sigma : \rho_\alpha(M) = \hat{\rho}(\alpha(M))$ where $\hat{\rho}(M)$ is the minimal deterministic automaton describing the language $\rho(\mathcal{L}(M))$. An abstraction α is *finitary*, if the set $\mathbb{A}_\Sigma$ is finite.

For a given abstraction function α, we can compute iteratively the sequence of automata $(\rho^i_\alpha(Init))_{i \geq 0}$. If $id \subseteq \rho$ (i.e., the identity is included in the transition relation), and the abstraction α is finitary (i.e., there exists just a finite number of automata that we can obtain after the abstraction), then there exists $k \geq 0$ such that $\rho_\alpha^{k+1}(Init) = \rho_\alpha^k(Init)$. The definition of the abstraction function α implies that $\mathcal{L}(\rho_\alpha^k(Init)) \supseteq \rho^*(\mathcal{L}(Init))$.

If $\mathcal{L}(\rho_\alpha^k(Init)) \cap \mathcal{L}(Bad) = \emptyset$, then the verification problem (1) has a positive answer. If the intersection is non-empty, we must check whether it is a real counterexample, or a spurious one. The spurious counterexample may be caused by the used abstraction (the counterexample is not reachable from the set of initial configurations). Assume that $\rho_\alpha^k(Init) \cap \mathcal{L}(Bad) \neq \emptyset$, which means that there is a symbolic path:

$$Init,\ \rho_\alpha(Init),\ \rho_\alpha^2(Init), \cdots \rho_\alpha^{n-1}(Init),\ \rho_\alpha^n(Init) \tag{2}$$

such that $\mathcal{L}(\rho_\alpha^n(Init)) \cap \mathcal{L}(Bad) \neq \emptyset$.

Let $X_n = \mathcal{L}(\rho_\alpha^n(Init)) \cap \mathcal{L}(Bad)$. Now, for each l, $0 \leq l < n$, we compute $X_l = \mathcal{L}(\rho_\alpha^l(Init)) \cap \rho^{-1}(X_{l+1})$. Two possibilities may occur: (*a*) $X_0 \neq \emptyset$, which means that the verification problem (1) has a negative answer, and $X_0 \subseteq \mathcal{L}(Init)$ is a set of dangerous initial configurations. (*b*) $\exists m, 0 \leq m < n : X_{m+1} \neq \emptyset \wedge X_m = \emptyset$ meaning that the abstraction function is too rough—we need to refine it and start the verification process again.

In [8], two general-purpose kinds of abstractions are proposed. Both are based on *automata state equivalences*. Tree automata states are split into several equivalence classes, and all states from one class are collapsed into one state. An abstraction becomes finitary if the number of equivalence classes is finite. The refinement is done by refining the equivalence classes. Both of the proposed abstractions allow for an automatic refinement to exclude the encountered spurious counterexample.

The first proposed abstraction is the *abstraction based on languages of trees of a finite height*. It defines two states equivalent if their languages up to a given height n are equal. There is just a finite number of languages of height n, therefore this abstraction is finitary. The abstraction may be refined by gradually increasing the height n. The second proposed abstraction is an *abstraction based on predicate languages*. Let $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ be a set of *predicates*. Each predicate $P \in \mathcal{P}$ is a tree language represented by a tree automaton. Let $M = (Q, \Sigma_C, F, \delta)$ be a tree automaton. Then, two states $q_1, q_2 \in Q$ are equivalent if their languages $\mathcal{L}(M, q_1)$ and $\mathcal{L}(M, q_2)$ have a nonempty intersection with exactly the same subset of predicates from the set $\mathcal{P}$. Since there is just a finite number of subsets of $\mathcal{P}$, the abstraction is finitary. The abstraction is refined by adding new predicates. In particular, if we take as new predicates the tree automata corresponding to the languages of all the states in the automaton of X_{m+1} from the analysis of a spurious counterexample ($X_m = \emptyset$), it can be shown [8] that the encountered spurious counterexample gets eliminated.

5.2 Using ARTMC in the Proposed Framework

Encoding of Configurations and Instructions. We use a slight modification of the basic ARTMC framework described above in that we do not work with a single set of reachable configurations. Instead, we compute a set of reachable configurations for each program line. Program lines may then be kept separately from the memory configurations (trees) reachable at these lines. The memory configurations may be encoded directly as trees from the set $\mathcal{T}(\mathcal{C})$ over the binary alphabet $\Sigma_C = Data \times 2^{PVar} \cup \{\square\}$—provided that we are given a program with pointer variables *PVar* and a set of tree data values *Data*. As explained already in Section 3, a tree node is either a leaf and represented by $\square$, or it contains a data value from the finite set *Data* and a set of pointer variables pointing to it. Each pointer variable can point to at most one tree node (if it is null, it does not appear in the tree).

Most of the program instructions that we consider can be encoded as structure-preserving tree transducers. Transducers can check conditions like $x == \texttt{null}$, $x == y$, or $x.\texttt{data} == d$ by simply checking whether x appears in some node of an input tree, whether x and y label the same tree node, or whether a node labelled with x is also labelled with the date item d, respectively. Transducers can also be used to move symbols representing the variables to nodes marked by some other variable ($x = y$), remove a symbol representing a variable from the tree ($x = \texttt{null}$), move it one level up or down ($\texttt{go}\{\texttt{Left}|\texttt{Right}|\texttt{Up}\}\texttt{NonNull}(x,y)$), or change the data element in the node marked by some variable from one value to another one ($\texttt{changeData}(x,d_1,d_2)$).

The instructions for tree rotations cannot be modelled by finite-state tree transducers. However, they (and their inverse) can still be expressed as regularity-preserving relations whose image can be effectively computed for any regular set using a special operation on tree automata. First, the test of the distribution of the program variables in the sets X,Y,A,B used in instructions $\{\texttt{left}|\texttt{right}\}\texttt{Rotate}(x,X,Y,A,B)$ can be easily implemented by an intersection with tree automata encoding the various possible mutual positions of the program variables.[4] The actual rotation is then the same for all the variants of the instruction. Let $M = (Q,\Sigma_C,F,\delta)$ be a tree automaton describing a set of of reachable configurations. The set of configurations reachable from $\mathcal{L}(M)$ via an instruction $\texttt{leftRotate}(x,X,Y,A,B)$ can be represented by a tree automaton M' that is obtained by the following algorithm (the case of $\texttt{rightRotate}(x,X,Y,A,B)$ being analogical):

- Start with $M' = M$.
- For all rules $c_x(q_1,q_2) \to q_t \in \Delta$ where $q_1,q_2,q_t \in Q$, $c_x \in \mathcal{C}$, and $x \in \nu(c_x)$ perform the following steps:
  - Remove rules of the form $\square \to q_2$ from Δ'—these rules (if present) cause the control to be taken to the special error state Err by the $\texttt{leftRotateErr}(x)$ instruction.
  - Remove the rule $c_x(q_1,q_2) \to q_t$ from Δ'.
  - For each rule $c_y(q_3,q_4) \to q_2 \in \Delta$ where $q_3,q_4 \in Q$ and $c_y \in \mathcal{C}$, do the following:
    * Choose a fresh state $q_m \notin Q'$.
    * Extend Δ' by the rules $c_x(q_1,q_3) \to q_m$ and $c_y(q_m,q_4) \to q_t$ and add q_m into Q'.

[4] As an optimization, one does not have to generate automata for all (exponentially many) theoretically possible distributions of the variables, but instead generate just automata for testing separately the position of each program variable wrt. the node pointed by x, intersect the current set of reachable configurations with the sets represented by these automata, and then use the obtained information to deduce which of the theoretical possibilities are feasible (in practice, there are typically only a few of them).

The idea of the algorithm is showed in Figure 6. Note that there can be several rules of the type $c_x(q_1,q_2) \to q_t$ with $x \in \nu(c_x)$ in the automaton M, but in each tree encoding a valid memory configuration there is at most one node pointed to by the variable x, and hence just one of these rules can be used in any accepting run. Therefore, by removing all of these rules, we safely break all the input trees just on the requested position. Then we can reconnect the parts of the broken trees by adding new rules. For each removed rule $c_x(q_1,q_2) \to q_t$, we find all rules of the type $c_y(q_3,q_4) \to q_2$, and for each of them, we add new rules $c_x(q_1,q_3) \to q_m$ and $c_y(q_m,q_4) \to q_t$ for a fresh state q_m. The remaining rules of the automaton are preserved, and so the parts of the trees around the rotation area are preserved. Also, note that all rules of the type $c_y(q_3,q_4) \to q_2$ must be preserved because they can be used outside of the rotation point too.

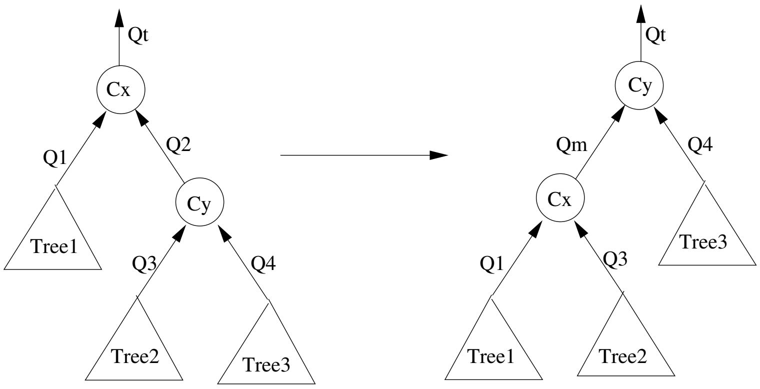

Figure 6: The left rotation on tree automata

The ARTMC Computation Loop and the ARTMC Abstractions Used. Since we compute a separate set of reachable configurations for each control point of the (possibly refined) CFG of the given program, we do not iterate a single transition relation as within the basic ARTMC framework mentioned above. Instead, we iteratively fire the instructions decorating arcs of the program CFG (i.e., we compute the image of the corresponding tree relations) whenever the set of configurations reachable at their source location changes. The computed image is then united with the already known set of reachable configurations for the appropriate target location and the obtained set (represented by a tree automaton) is abstracted.

Out of the abstractions commonly used in the ARTMC framework, we can use both the abstraction based on predicate languages as well as the finite-height abstraction in the initial generation of reachable configurations. These two abstractions can also be combined in such a way that two automata states are equal when they satisfy the same predicate languages and accept the same trees up to some height. In the later stages of our termination checking loop from Figure 5, we always use a gradually refined finite height abstraction, which may be combined with the predicate abstraction. However, we never use the predicate-based abstraction alone because its refinement schema requires one to have finite (not lasso-shaped) counterexamples.

6 Abstraction of Programs with Trees into Counter Automata

In this section, we provide a translation from tree manipulating programs to counter automata such that existing techniques for proving termination of counter automata can be used to prove termination of the programs. Before describing the translation, we define the simulation notion that we will use to formalize correctness of the translation.

Let P be a program with a set of instructions $Instr$, an initial label $l_0 \in Lab$, a set of input trees $I_0 \subseteq \mathcal{T}(\mathcal{C})$, and a set of reachable configurations $\mathcal{R}_P \subseteq Lab \times \mathcal{T}(\mathcal{C})$. Let us also have a counter automaton $A = \langle X, Q, q_0, \varphi_0, \rightarrow \rangle$ with $\rightarrow \in Q \times \Phi(X) \times Q$, and a set of reachable configurations $\mathfrak{R}_A$. A function $M : X \times \mathcal{T}(\mathcal{C}) \rightarrow \mathbb{Z}$ is said to be a *measure* assigning counters integer values for a particular tree[5]. Let $\mathbf{M}(t) = \{M(x,t) \mid x \in X\}$.

Definition 1 *The program P is simulated by the counter automaton A w.r.t. $M : X \times \mathcal{T}(\mathcal{C}) \rightarrow \mathbb{Z}$ and $\theta : Instr \rightarrow \Phi$ iff there exists a relation $\sim \subseteq \mathcal{R}_P \times \mathfrak{R}_A$ such that (1) $\forall t_0 \in I_0 : \mathbf{M}(t_0) \models \varphi_0 \wedge \langle l_0, t_0 \rangle \sim \langle q_0, \mathbf{M}(t_0) \rangle$ and (2) $\forall (l_1,t_1),(l_2,t_2) \in \mathcal{R}_P \; \forall i \in Instr \; \forall (q_1, \nu_1) \in \mathfrak{R}_A : (l_1,t_1) \xrightarrow{i} (l_2,t_2) \wedge (l_1,t_1) \sim (q_1,\nu_1) \Rightarrow \exists (q_2,\nu_2) \in \mathfrak{R}_A : (q_1,\nu_1) \xrightarrow{\theta(i)} (q_2,\nu_2) \wedge (l_2,t_2) \sim (q_2,\nu_2)$.*

The measure M ensures that the counters are initially correctly interpreted over the input trees, whereas θ ensures that the counters are updated in accordance with the manipulations done on the trees. Simulation in the sense of Definition 1 guarantees that if we prove termination of the CA, the program will terminate on any $t \in I_0$.

6.1 Abstract Control Flow Graphs

According to Figure 4, we construct the CA simulating a program in two steps: we first construct the so-called *abstract control flow graph* (ACFG) of a program, and then translate it into a CA. Initially, the ACFG of a program is computed from its CFG by decorating its nodes with ARTMC-overapproximated sets of configurations reachable at each line (we keep the initial set of trees exact exploiting the fact that w.l.o.g. there are no statements leading back to the initial line). These sets allow us to exclude impossible (not fireable) transitions from the ACFG and thus derive a more exact CA. Further, in subsequent refinement iterations, infeasible termination counterexamples are excluded by splitting these sets (if this appears to be insufficient, we re-run ARTMC to compute a better overapproximation of the reachable sets of configurations). Below, we first define the notion of ACFG, then we provide its translation to counter automata.

In what follows, let P be a program (pre-processed according to Section 3.1) with instructions $Instr$, working on trees from $\mathcal{T}(\mathcal{C})$, and let $l_0 \in Lab$ be the initial line of P. The *control flow graph* (CFG) of P is a labelled graph $F = \langle Instr, Lab, l_0, \Rightarrow \rangle$ where $l \overset{i}{\Rightarrow} l'$ denotes the presence of an instruction i between control locations $l, l' \in Lab$. We further suppose that the input tree configurations for P are described by the user as a (regular) set of trees $I_0 \subseteq \mathcal{T}(\mathcal{C})$. An *abstract*

[5] Intuitively, certain counters will measure, e.g., the distance of a certain node from the root, the size of the subtree below it, etc.

control flow graph (ACFG) for P is then a graph $G = \langle Instr, LI, \langle l_0, I_0\rangle, \longmapsto\rangle$ where LI is a *finite* subset of $Lab \times 2^{\mathcal{T}(\mathcal{C})}$, $\langle l_0, I_0\rangle \in LI$, and there is an edge $\langle l,I\rangle \overset{i}{\longmapsto} \langle l',I'\rangle$ iff $l \overset{i}{\Rightarrow} l'$ in the CFG of P and $post(i,I) \cap I' \neq \emptyset$. For a finite sequence of instructions $\pi : i_1 i_2 \ldots i_n$, we denote by $l \overset{\pi}{\longmapsto} l'$ the existence of a path labelled with π between l and l' in G.

Note that as we always work with ACFGs annotated with regular sets of configurations and we can implement the effect of each instruction on a regular set as an operation on tree automata, we can easily implement the test $post(i,I) \cap I' \neq \emptyset$ needed for computing the edges of ACFGs. As an optimization of introducing edges corresponding to $\{$`left`$|$`right`$\}$`Rotate`(x,X,Y,A,B), we might first test for each pointer variable in which of the subtrees in Figure 2 it appears relative to the node pointed by x (which can be tested by an intersection with simple tree automata encoding the various mutual positions). Then, we take into account only the sets X,Y,A,B based on these possibilities (which in practice are typically only a few) rather than blindly testing all (exponentially many) theoretically possible variants of the distribution of the variables in X,Y,A,B. Note also that a location in P may correspond to more than one locations in G.

We say that G *covers the invariants of* P whose set of reachable states is $\mathcal{R}_P$ iff each tree $t \in \mathcal{T}(\mathcal{C})$ that is reachable at a program line $l \in Lab$ (i.e., $\langle l,t\rangle \in \mathcal{R}_P$), appears in some of the sets of program configurations associated with l in the locations of G. Formally, $\forall l \in Lab$: $\mathcal{R}_P \cap (\{l\} \times \mathcal{T}(\mathcal{C})) \subseteq \{l\} \times \bigcup_{\langle l,I\rangle \in LI} I$. The following lemma captures the relation between the semantics of a program and that of an ACFG.

Lemma 1 *Let P be a program with trees and G an ACFG that covers the invariants of P. Then, the semantics of G simulates that of P in the classical sense.*

Proof: We define a relation $\preceq \subseteq \mathcal{R}_P \times ((Lab \times 2^{\mathcal{T}(\mathcal{C})}) \times \mathcal{T}(\mathcal{C}))$ between reachable configurations of the program and configurations of the ACFG as follows. $(l,t) \preceq (\langle l_1, I\rangle, t_1)$ iff $l = l_1$, $t = t_1$ and $t_1 \in I_1$. We show that $\preceq$ is a simulation relation. Let $(l,t) \preceq (\langle l,I\rangle, t)$ and $(l,t) \xrightarrow[P]{i} (l',t')$. Then, since $(l',t') \in \mathcal{R}_P$ and G covers P, there exists $\langle l',I'\rangle \in LI$ with $t' \in I'$ and furthermore $\langle l,I\rangle \overset{i}{\longmapsto} \langle l',I'\rangle$ as $post(i,I) \cap I' \neq \emptyset$. We have $(l',t') \preceq (\langle l',I'\rangle, t')$. □

For illustration, in Figures 7 and 8, we present the CFG and the initial ACFG of the DFT program from Figure 1(b) obtained by labelling the DFT by invariants computed by ARTMC. The invariants are depicted such that the shaded part represents the (possibly empty) part of a tree with only marked nodes, the white part represents the (possibly empty) part of a tree with only unmarked nodes, and the part with the question mark represents the part of the tree where the nodes may be both marked as well as unmarked. The upper circle represents the node pointed by the variable x (if it is white, the node is unmarked, if it is black, the node is marked), and the lower circle represents the node pointed by the variable y. There is just one circle in the cases where $x == y$ or $y ==$ `null`. These circles represent the relative positions of the nodes pointed by the variables x and y wrt. the marked/unmarked areas of the tree.

Note that all arcs leading to the error location Err as well one of the arcs between nodes 10 and 11 from the CFG are missing in the ACFG. This is a consequence of using the invariants. In fact, a removal of the arc between nodes 10 and 11 will later lead to a CA allowing us to show that DFT terminates.

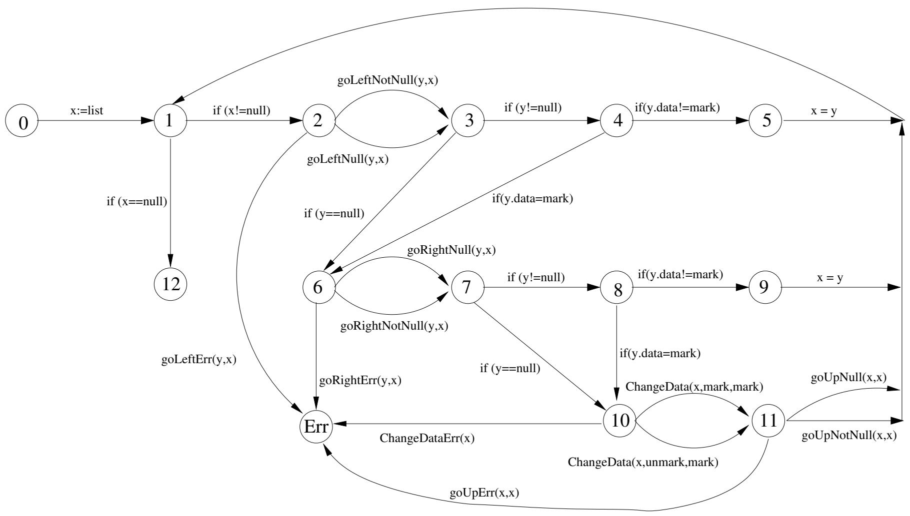

Figure 7: The CFG of the DFT procedure from Figure 1(b)

6.2 Translation to Counter Automata

We now describe the construction of a CA $A_{rsc}(G) = \langle X, Q, q_0, \varphi_0, \rightarrow \rangle$ from an ACFG $G = \langle Instr, LI, \langle l_0, I_0 \rangle, \longmapsto \rangle$ of a program P such that $A_{rsc}(G)$ simulates P in the sense of Def. 1. We consider two sorts of counters, i.e., $X = X_{PVar} \cup X_{Data}$ where $X_{PVar} = \{r_x \mid x \in PVar\} \cup \{s_x \mid x \in PVar\}$ and $X_{Data} = \{c_d \mid d \in Data\}$. For each $x \in PVar$, the counter r_x keeps track of the current distance of the node n pointed by x from the root of t, whereas the counter s_x keeps track of the size of the subtree below n (with both r_x and s_x being set to -1 iff x is null). For each $d \in Data$, c_d is contains the number of d-valued nodes of t. Formally, using the notion of measures, we define for each $x \in PVar$ that $M_{rsc}(r_x, t) = |p|$ and $M_{rsc}(s_x, t) = |dom(t_{|p})|$ if there exists $p \in dom(t)$ such that $x \in \nu(t(p))$, and $M_{rsc}(r_x, t) = M_{rsc}(s_x, t) = -1$ otherwise. For each $d \in Data$, we define $M_{rsc}(c_d, t) = |\{p' \in dom(t) \mid \delta(t(p')) = d\}|$.

We build $A_{rsc}(G)$ from G by simply replacing the instructions on edges of G by operations on counters. Formally, this is done by the translation function θ_{rsc} defined in Table 1. The mapping for the instructions $x = y$.`right` and `rightRotate`(x, X, Y, A, B) is skipped in Table 1 as it is analogous to that of $x = y$.`left` and `leftRotate`(x, X, Y, A, B), respectively. Also, for brevity, we skip the instructions leading to the error state Err. As a convention, if the future value of a counter is not explicitly defined, we consider that the future value stays the same as the current value. Moreover, in all formulae, we assume an implicit guard $-1 \leq r_x < TreeHeight \wedge -1 \leq s_x < TreeSize$ for each $x \in PVar$[6] and $0 \leq c_d \leq TreeSize \wedge \sum_{d \in Data} c_d = TreeSize$ for each $d \in Data$. $TreeHeight$ and $TreeSize$ are parameters restricting the range in which the other counters can change according to a given input tree. They are needed as a basis on which termination of the resulting automaton can be shown.

[6] -1 corresponds to x being null.

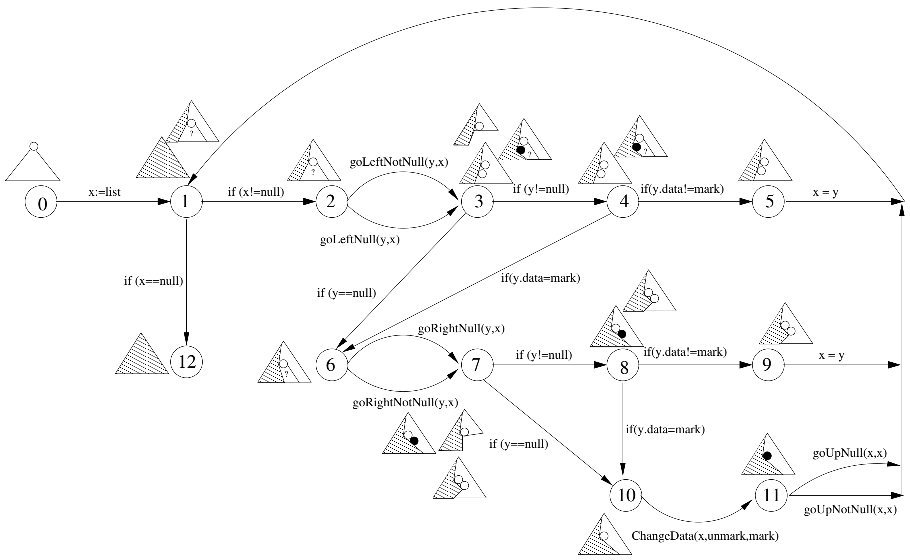

Figure 8: The initial ACFG of the DFT procedure from Figure 1(b)

Next, we define $Q = LI$, $q_0 = \langle l_0, I_0 \rangle$, and $q \xrightarrow{\theta_{rsc}(i)} q'$ iff $q \overset{i}{\longmapsto} q'$ for all $i \in Instr$. The initial constraint φ_0 on the chosen counters can be automatically computed[7] from the regular set of input trees I_0 such that it satisfies requirement (1) of Definition 1. The following theorem shows the needed simulation relation between the counter automata we construct and the programs.

To illustrate the construction of counter automata from ACFGs, we present in Figure 9 the counter automaton built from the ACFG from Figure 8 of our running example, i.e., the DFT procedure from Figure 1(b).

Theorem 2 *Given a program P and an ACFG G of P covering its invariants, the CA $A_{rsc}(G)$ simulates P in the sense of Definition 1 wrt. θ_{rsc} and M_{rsc}.*

The counter automata $A_{rsc}(G)$ that we generate have further the property that each transition $q \xrightarrow{\varphi} q'$ can be mapped back into the program instruction from which it originates. This is because the instructions onto which we decompose each program statement are assigned different formulae, by the translation function θ_{rsc}, and there is at most one statement between each two control locations of the program. Formally, we capture this by a function $\xi : Q \times \Phi \times Q \to Instr$ such that $\forall q_1, q_2 \in Q, \varphi \in \Phi : q \xrightarrow{\varphi} q' \Rightarrow q \overset{\xi(q_1,\varphi,q_2)}{\longmapsto} q'$. We generalize θ_{rsc} and ξ to sequences of transitions, i.e., for a path π in A_{rsc}, $\xi(\pi)$ denotes the sequence of program instructions leading to π, and $\theta_{rsc}(\xi(\pi))$ denotes the sequence of counter operations on π obtained by projecting out the control locations from π.

[7]This can be done by computing the Parikh image of a context-free language $\mathcal{L}(I_0)$ corresponding to the regular tree language I_0. For each tree $t \in I_0$ there is a word in $\mathcal{L}(I_0)$ consisting of all nodes of t. We use special symbols to denote the position of a node in the tree relative to a given variable (under the variable, between it and the root) and the data values of nodes.

Table 1: The mapping θ_{rsc} from program instructions to counter manipulations

instruction i	counter manipulation $\theta_{rsc}(i)$
$\mathtt{if}(x == \mathtt{null})$	$r_x = -1$
$\mathtt{if}(x! = \mathtt{null})$	$r_x \geq 0$
$\mathtt{if}(x == y)$	$r_x = r_y \wedge s_x = s_y$
$\mathtt{if}(x! = y)$	$true$
$\mathtt{if}(x.\mathtt{data} == d)$	$r_x \geq 0 \wedge c_d \geq 1$
$\mathtt{if}(x.\mathtt{data}! = d)$	$r_x \geq 0 \wedge c_d < TreeSize$
$x = \mathtt{null}$	$r'_x = s'_x = -1$
$x = y$	$r'_x = r_y \wedge s'_x = s_y$
$\mathtt{goLeftNull}(x,y)$	$r_y \geq 0 \wedge s_y \geq 1 \wedge r'_x = s'_x = -1$
$\mathtt{goLeftNonNull}(x,y)$	$r_y \geq 0 \wedge s_y \geq 2 \wedge r'_x = r_y + 1 \wedge s'_x < s_y$
$\mathtt{goUpNull}(x,y)$	$r_y = 0 \wedge s_y \geq 1 \wedge r'_x = s'_x = -1$
$\mathtt{goUpNonNull}(x,y)$	$r_y \geq 1 \wedge s_y \geq 1 \wedge r'_x = r_y - 1 \wedge s'_x > s_y$
$\mathtt{changeData}(x,d,d)$	$r_x \geq 0 \wedge s_x \geq 1 \wedge c_d > 0$
$\mathtt{changeData}(x,d_1,d_2), d_1 \neq d_2$	$r_x \geq 0 \wedge s_x \geq 1 \wedge c_{d_1} > 0 \wedge c'_{d_2} = c_{d_2} + 1 \wedge c'_{d_1} = c_{d_1} - 1$
$\mathtt{leftRotate}(x,X,Y,A,B)$	$gLeftRotate(x,X,Y,A,B) \wedge aLeftRotate(x,X,Y,A,B)$

$$\begin{array}{l}
gLeftRotate(x,X,Y,A,B) = \\
\quad r_x \geq 0 \ \wedge \ s_x \geq 2 \ \wedge \\
\quad (\forall v \in X : r_v = r_x \ \wedge \ s_v = s_x) \ \wedge \\
\quad (\forall v, v' \in Y : r_v = r_x + 1 \ \wedge \ s_v < s_x \ \wedge \\
\qquad\qquad r_v = r_{v'} \ \wedge \ s_v = s_{v'}) \ \wedge \\
\quad (\forall v \in A : r_v \geq r_x + 1 \ \wedge \ s_v < s_x) \ \wedge \\
\quad (\forall v \in B : r_v \geq r_x + 2 \ \wedge \ s_v < s_x - 1)
\end{array}
\qquad
\begin{array}{l}
aLeftRotate(x,X,Y,A,B) = \\
\\
\quad (\forall v \in X : r'_v = r_v + 1 \ \wedge \ s'_v < s_v) \ \wedge \\
\quad (\forall v \in Y : r'_v = r_v - 1 \ \wedge \ s'_v > s_v) \ \wedge \\
\\
\quad (\forall v \in A : r'_v = r_v + 1 \ \wedge \ s'_v = s_v) \ \wedge \\
\quad (\forall v \in B : r'_v = r_v - 1 \ \wedge \ s'_v = s_v)
\end{array}$$

7 Checking Spuriousness of Counterexamples

Since the CA A_{rsc} generated from a program P with trees is a simulation of P (cf. Theorem 2), proving termination of A_{rsc} suffices to prove termination of P. However, if A_{rsc} is not proved to terminate by the termination checker of choice, there are three possibilities: (1) A_{rsc} terminates, but the chosen termination checker did not find a termination argument, (2) both A_{rsc} as well as P do not terminate, and (3) P terminates, but A_{rsc} does not, as a consequence of the abstraction used in its construction. In all these three cases, the CA termination checker outputs a counterexample consisting of a finite path (stem) that leads to a cycle, both paths forming a lasso.

Formally, given a CA $A = \langle X, Q, q_0, \varphi_0 \rangle$, a lasso is a path in the control graph of A of the form $S.L$ where $S = q_0 \xrightarrow{\varphi_1^s} q_1^s \xrightarrow{\varphi_2^s} q_2^s \xrightarrow{\varphi_3^s} \ldots q_{m-1}^s \xrightarrow{\varphi_m^s} q_1^l$ and $L = q_1^l \xrightarrow{\varphi_1^l} q_2^l \xrightarrow{\varphi_2^l} q_3^l \xrightarrow{\varphi_3^l} \ldots q_n^l \xrightarrow{\varphi_n^l} q_1^l$.

The lasso is said to be *spurious* if and only if there exists a non-terminating run $\langle q_0, \nu_0 \rangle \xrightarrow{\varphi_1^s}$

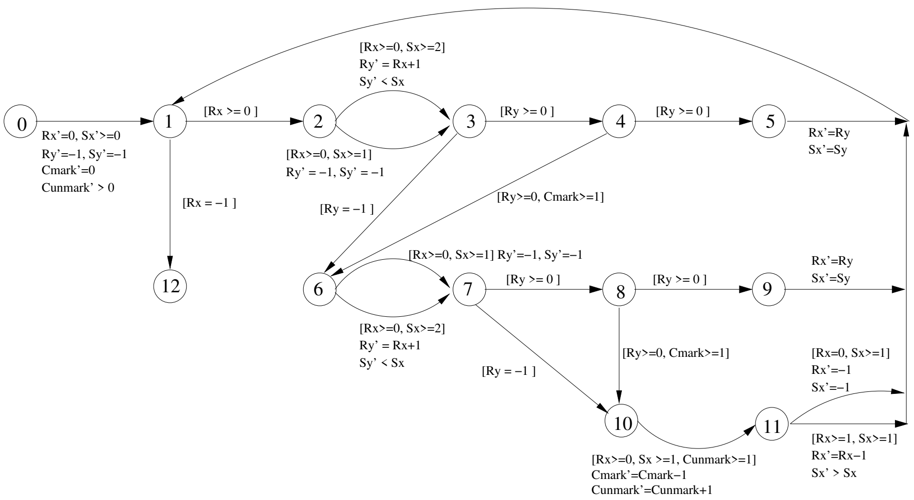

Figure 9: The CA built from the ACFG from Figure 8 of the DFT program from Figure 1(b)

$\ldots \langle q^s_{m-1}, \nu_{m-1}\rangle \xrightarrow{\varphi^s_m} \left(\langle q^l_1, \nu_m\rangle \xrightarrow{\varphi^l_1} \ldots \xrightarrow{\varphi^l_n} \langle q^l_1, \nu_m\rangle\right)^\omega$ in A where $\nu_0 \models \varphi_0$, and P does not have an infinite run along the path $\xi(q_0, \varphi^s_1, q^s_1) \ldots \xi(q^s_{m-1}, \varphi^s_{m-1}, q^l_1)\left(\xi(q^l_1, \varphi^l_1, q^l_2) \ldots \xi(q^l_n, \varphi^l_n, q^l_1)\right)^\omega$ for any $t \in I_0$.

The three above mentioned cases that may arise when we are given a lasso shaped counterexample over the generated CA are dealt with in the upcoming paragraphs.

Deciding termination of CA lassos. We first show that termination of a given control loop is decidable in a CA whose transition relations are conjunctions of formulae of the forms $x - y \leq c$, $x' - y \leq c$, $x - y' \leq c$, or $x' - y' \leq c$ where x' denotes the future value of the counter x and $c \in \mathbb{Z}$ is an arbitrary integer constant. For this, we exploit the results of [16, 12]. It is clear that the CA generated via the translation function θ_{rsc} fall into this class. In particular, the constraint that each counter is bounded from below by -1 and from above by the $TreeHeight$ or $TreeSize$ parameters is expressible using affine transitions.[8]

Theorem 3 *Let $A = \langle X, Q, q_0, \varphi_0, \rightarrow\rangle$ be a counter automaton with affine transition relations. Then, given a control loop in A, the problem whether there exists an infinite computation along the loop is decidable.*

Proof: It is well-known that the class of affine relations is closed under composition, defined as $(R_1 \circ R_2)(\mathbf{x}, \mathbf{x}') = \exists \mathbf{y} \,.\, R_1(\mathbf{x}, \mathbf{y}) \wedge R_2(\mathbf{y}, \mathbf{x}')$. In other words, the existential quantifiers can be eliminated[9], the result being written as another affine relation. As a consequence, we can assume w.l.o.g. that the lasso consists of only one affine transition, together with some initial condition

[8]To encode conditions of the form $x \leq c$ we add a new variable z, initially set to zero, with the condition $z' = z$ appended to each transition, and rewrite the original condition as $x - z \leq c$.

[9]By, e.g., the Fourier-Motzkin procedure.

that summarizes the stem. In general, an affine relation $R(\mathbf{x}, \mathbf{x}')$ can be represented as a directed weighted graph whose set of vertices is the set of variables $\mathbf{x} \cup \mathbf{x}'$, and there is an edge with weight $c \in \mathbb{Z}$ from x to y if and only if there is an explicit constraint $x - y \leq c$ in φ. An n-step iteration of the loop is represented by a *constraint graph* G_n, defined as the minimal graph whose set of vertices is $\bigcup_{i=0}^{n} \mathbf{x}^i$, where $\mathbf{x}^i = \{y^i \mid y \in \mathbf{x}\}$ and, for all $0 \leq i < n$, there is an edge labeled c:

- from x^i to y^i, if there is a constraint $x - y \leq c$ in R.
- from x^{i+1} to y^{i+1}, if there is a constraint $x' - y' \leq c$ in R.
- from x^i to y^{i+1}, if there is a constraint $x - y' \leq c$ in R.
- from x^{i+1} to y^i, if there is a constraint $x' - y \leq c$ in R.

Intuitively, the nodes $\mathbf{x^i}$ in G_n represent the possible values of the counters after i iterations. If $\pi : x^i \xrightarrow{c_1} \ldots \xrightarrow{c_m} y^j$, $0 \leq i, j, \leq n$ is a path in G_n, let $\omega(\pi)$ denote the sum of all labels along the path, i.e. $\omega(\pi) = \sum_{k=1}^{m} c_k$. Clearly, we have $x^i - y^j \leq \omega(\pi)$, and moreover, this is the strongest relation involving the values of x and y at the instants i and j, respectively. Therefore, for any $n > 0$, there exists a run along the loop, consisting of at least n transitions, if and only if all constraints in G_n are satisfiable. The last condition is equivalent to the absence of negative cycles in G_n. One of the results from [16, 12] is that the existence of a negative cycle in G_n, for any n can be encoded by a formula in Presburger arithmetic. Consequently, the existence of an infinite run along the loop is also Presburger definable, and thus decidable. □

Checking termination of program lassos. Due to the above result, we may henceforth assume that the lasso $S.L$ returned by the termination analyzer has a real non-terminating run in the CA. The lasso is mapped back into a sequence of program instructions $\xi(S).\xi(L)$ forming a program lasso. Two cases may arise: either the lasso is real on the program, or it is spurious.

Non-spurious program lassos. Since we do not consider dynamic allocation, the number of configurations that a program can reach from any input tree is finite. Consequently, if there is a tree t_ω from which the program will have an infinite run along a given lasso, then we can discover it by an exhaustive enumeration of trees. Moreover, as shown in the innermost loop of the analysis procedure in Section 4, using the symbolic implementation of the program instructions over tree automata, we can handle the discovery of t_ω by evaluating the lasso symbolically for all trees of a certain (increasingly growing) height at the same time. As we work with finite sets of trees, we are bound to visit the same set twice after a finite number of iterations if there exists a non-terminating run along the lasso.

Note that, unfortunately, we cannot replace the above approach of examining gradually trees of increasing size by a single symbolic run starting with I_0. The set I_0 may contain both trees for which the program loops and also infinitely many trees for which infinite looping is impossible, but at the same time, the number of times the loop of the lasso can be fired for the latter trees

```
while (x ≠ root) do
  leftRotate(x);
  x := x.up;
  y := y.up;
  rightRotate(x);
  x := x.up;
  y := y.up;
```

Figure 10: A possibly nonterminating program

depends on their size (and hence is not fixed). Then, if we try to symbolically run the lasso starting from all input trees, we will never converge as in every iteration of the loop, the working set of trees will shrink by some trees for which a further iteration is not possible. This situation happens, e.g., in the program in Figure 10 where if we initially have $x == y$, the run will not stop (provided the node pointed by x has a right successor). On the other hand, if y points to a node above the node pointed by x, the program will stop, but after a number of steps depending on the initial distance of y from the root.

Spurious program lassos. We handle this case also by a symbolic iteration of a given program lasso $\sigma.\lambda$ starting with the initial set of trees. We compute iteratively the sets $post(\sigma.\lambda^k, I_0), k = 1, 2, \ldots$. In the case of lassos without destructive updates, this computation is shown to reach the empty set after a number of iterations that is bounded by a double exponential in the length of the lasso (cf. Section 7.1). In the case of lassos with destructive updates, we can guarantee termination of the iteration with the empty set provided there exists some CA A_u (albeit unknown) keeping track of the particular tree measures we consider here (formalized via the functions M_{rsc} and θ_{rsc} in Section 6.2) that simulates the given program and that terminates[10] (cf. Section 7.2). In the latter case, even though we cannot guarantee the discovery of A_u, we can at least ensure that the sequence $post(\sigma.\lambda^k, I_0)$, $k = 1, 2, \ldots$ terminates with the empty set. This gives us a basis for refining the current ACFG such that we get rid of the spurious lasso encountered, and we can go on in the search for a CA showing the termination of the given program.

7.1 Deciding Spuriousness of Lassos without Destructive Updates

In this section, we show that the spuriousness problem for a given lasso in a program with trees is decidable, if the lasso does not contain destructive updating instructions, i.e., tree rotations. The argument for decidability is that if there exists a non-terminating run along the loop, then there exists also a non-terminating run starting from a tree of size bounded by a constant depending on the program. Thus, there exists a tree within this bound that will be visited infinitely many often.[11]

[10] We can relax this condition by saying that A_u does not have any infinite run, not corresponding to a run of the program. For the sake of clarity, we have chosen the first stronger condition.

[11] Since there is no dynamic allocation, all trees visited starting with a tree of size k will also have size k. Hence each run of the program will either stop, or re-visit the same program configuration after a bounded number of steps.

Given a loop without destructive updates, we first build an abstraction of it by replacing the go{Left|Right|Up}Null(x,y) instructions by $x = \texttt{null}$, and by eliminating all instructions changeData(x,d_1,d_2) and the tests. Clearly, if the original loop has a non-terminating computation, then its abstraction will also have a non-terminating run starting with the same tree. The loop is then encoded as an iterative linear transformation which, for each pointer variable $x \in PVar$, has a counter p_x encoding the binary position of the pointer in the current tree using 0/1 as the left/right directions. Additionally, the most significant bit of the encoding is required to be always one, which allows for differentiating between, e.g., the position 001 encoded by $9 = (1001)_2$, and 0001 encoded by $17 = (10001)_2$. Null pointers are encoded by the value 0. The program instructions are translated into counter operations as follows:

$$x = \texttt{null} : p_x = 0 \qquad x = y : p_x = p_y \qquad \texttt{goLeftNonNull}(x,y) : p_x = 2\star p_y$$
$$\texttt{goRightNonNull}(x,y) : p_x = 2\star p_y + 1 \qquad \texttt{goUpNonNull}(x,y) : p_x = \tfrac{1}{2} p_y$$

where $2\star$ and $\frac{1}{2}$ denote the integer functions of multiplication ($x \mapsto 2x$) and division ($x \mapsto x/2$). Assuming that we have n pointer variables, each program instruction is modeled by a linear transformation of the form $\mathbf{p}' = A\mathbf{p} + B$ where A is an $n \times n$ matrix with at most one non-null element, which is either $1, 2$ or $\frac{1}{2}$, and B is an n-column vector with at most one 1 and the rest 0[12]. The composition of the instructions on the loop is also a linear transformation, except that A has at most one non-null element on each line, which is either I, or a composition of $2\star$'s and $\frac{1}{2}$'s.

Since A has at most one non-null element on each line, one can extract an $m \times m$ matrix A_0 for some $m \leq n$ that has exactly one non-null element on each line and column. Our proof is based on the fact that there exists some constant k bounded by $O(3^m)$ such that ${A_0}^k$ is a diagonal matrix. Intuitively, this means that the position of each pointer at step $i+k$ is given by a linear function of the position of the pointer at i. Then A^i is an exponential function of i. As there is no dynamic allocation of nodes in the tree, the non-termination hypothesis implies that the positions of pointers have to stay in-between bounds. But this is only possible if the elements of the main diagonal of ${A_0}^k$ are either I or compositions of the same number of $2\star$ and $\frac{1}{2}$. Intuitively, this means that all pointers are confined to move inside bounded regions of the working tree.

Theorem 4 *Let P be a program over trees, $PVar$ and $Data$ be its sets of pointer variables and data elements, $\mathcal{C} = Data \times 2^{PVar}$, $I_0 \subseteq \mathcal{T}(\mathcal{C})$ be an initial set of trees, and $\sigma.\lambda$ be a lasso of P. Then, if P has an infinite run along the path $\sigma.\lambda^\omega$ for some $t_0 \in I_0$, then there exists a tree $t_{b0} \in \mathcal{T}(\mathcal{C})$ of height bounded by $(\|PVar\|+1) \cdot \left(|\sigma| + |\lambda| \cdot 3^{\|PVar\|}\right)$ such that P, started with t_{b0}, has an infinite run along the same path.*

Proof: We assume that the loop λ of the lasso consists of at least one instruction of the form go{Left|Right|Up} NonNull(x,y), otherwise the set of positions of all pointer values will remain the same throughout the run, and non-termination can be trivially witnessed by a tree of height (and also size) bounded by $\|PVar\|$. This is because all guards of λ are of the form $(x = y)$, $(x = \text{null})$, $(x.\text{data} = d)$, and boolean combinations of the above. These guards partition the set

[12] We interpret the matrix operations over the semiring of integer functions $\langle \mathbb{N} \to \mathbb{N}, +, \circ, 0, I\rangle$ where $\circ$ is functional composition and I is the identity function.

of tree positions into at most $\|PVar\|+1$ equivalence classes[13]. If λ has an infinite run, then all of them have to be satisfied by a given tree, whose size is bounded by $\|PVar\|+1$.

First, we abstract the lasso $\sigma.\lambda$ into a lasso $\sigma.\lambda_a$, such that the loop λ_a consists only of $x = \texttt{null}$, $x = y$ and $\texttt{go}\{\texttt{Left}|\texttt{Right}|\texttt{Up}\}\texttt{NonNull}(x,y)$ instructions. This is done by replacing all guards with $\texttt{if}(\texttt{true})$, $\texttt{go}\{\texttt{Left}|\texttt{Right}|\texttt{Up}\}\texttt{Null}(x,y)$ instructions by $x = \texttt{null}$, and eliminating all $\texttt{changeData}(x,d_1,d_2)$ instructions. It can be shown that, if there exists a non-terminating run following the path $\sigma.\lambda^i.\left(\lambda^j\right)^\omega$, for some $i,j>0$, then there exists also a non-terminating run following the path $\sigma.\lambda^i.\left(\lambda_a^j\right)^\omega$. Note that $|\lambda_a| \leq |\lambda|$.

Second, we encode the loop λ_a as a linear transformation $\mathbf{p}' = A\mathbf{p}+B$ where $\mathbf{p},\mathbf{p}'$ and B are column vectors of size $\|PVar\|$, and A is a $\|PVar\| \times \|PVar\|$ matrix. As previously discussed, each component of $\mathbf{p}$ encodes the current position of a pointer variable, A has at most one non-zero element on each line, and this is a composition of $2\star$ and $\frac{1}{2}$, and B is composed only of 0 and positive integer values.

The condition of a non-terminating run along the path $\sigma.\lambda^i.\left(\lambda_a^j\right)^\omega$ implies that all pointer variables occurring in an instruction of the form $\texttt{go}\{\texttt{Left}|\texttt{Right}|\texttt{Up}\}\texttt{NonNull}(x,y)$ have to be non-null infinitely often, otherwise one of the guards would be violated and the instruction would not be executable. This induces the following condition on the counter variables from $\mathbf{p}$: for each $x \in PVar$ that occurs in an $\texttt{go}\{\texttt{Left}|\texttt{Right}|\texttt{Up}\}\texttt{NonNull}(x,y)$ instruction, the value of p_x (1) has to be infinitely often non-zero and (2) may not increase indefinitely.

The property of A of having at most one non-zero element on each line can be formalized as a partial mapping $nz : \{1,\ldots,\|PVar\|\} \rightarrow \{1,\ldots,\|PVar\|,\perp\}$ where for a line number $1 \leq i \leq \|PVar\|$, $nz(i)$ is the position of the non-zero element, if any, or $\perp$ otherwise. Let A_0 be the submatrix of A, and B_0 be a subvector of B, obtained in the following way:

- for all $1 \leq i \leq \|PVar\|$ if line i of A has only zero elements, then cross out line i, as well as column i of A, and line i of B.
- for each line $i = 1,2,\ldots,\|PVar\|$ cross out all lines and columns of A $j > i$, with $nz(j) = nz(i)$, as well as each line j of B.

The resulting matrix A_0 has exactly one non-zero element on each line and each column. The intuition behind this is to transform the initial system $\mathbf{p}' = A\mathbf{p}+B$ into a system $\mathbf{p_0'} = A_0\mathbf{p_0}+B_0$ in which the next value of each variable is computed from the value of another variable, no variable has a constant value, and no two variables are computed based on the same variable. Note that, for all counters $p_x \in \mathbf{p_0}$ that occur in the new system, the corresponding pointer variables $x \in PVar$ have to occur within either $\texttt{go}\{\texttt{Left}|\texttt{Right}|\texttt{Up}\}\texttt{NonNull}(x,y)$ or $x = y$ instructions.

In analogy with a permutation matrix, there exists a constant k such that ${A_0}^k$ is a diagonal matrix[14]. This constant is bounded by $3^m \leq 3^{\|PVar\|}$ where m is the size of A_0[15]. Moreover, the elements on the main diagonal of ${A_0}^k$ are (at most k times) compositions of $2\star$'s and $\frac{1}{2}$'s.

[13] Note that guards of the form $(x.\text{data} = d)$ may not introduce more classes in this partition due to the relation between pointer and data equality, namely $x = y \Rightarrow x.\text{data} = y.\text{data}$.

[14] If A_0 had only 0 and 1 elements, with exactly only 1 on each line and column, it would represent a permutation, and the constant k would be the permutation order. The behavior of A_0 is similar to the one of a permutation matrix, except the fact that, at each iteration the non-zero values of A_0^i are products of its non-zero elements.

[15] The permutation order is the least common multiple of the sizes of all cyclic permutations, which sum up to

Consequently, we have $A_0^n = \left(A_0^k\right)^{n/k} \cdot A_0^{n \mod k}$, for any $n \geq 0$, i.e. $\left(A_0^n\right)_{ij} = \left(\left(A_0{}^k\right)_{ii}\right)^{n/k} \cdot \left(A_0^{n \mod k}\right)_{ij}$ for all $1 \leq i,j \leq m$. Having obtained a closed-form expression for A_0^n, allows us to express the values of each p_x, $x \in PVar$ as functions of the form:

$$\sum_{i=1}^{m} \left(\left(A_0{}^k\right)_{ii}\right)^{n/k} \alpha_i(n \mod k)$$

where α_i are positive coefficients depending on $n \mod k$.

In order to meet conditions (1) and (2) it must be that $\left(A_0{}^k\right)_{ii}$ is a compositions of as many $2\star$'s as $\frac{1}{2}$'s. For assume that the number of $2\star$'s is bigger than the number of $\frac{1}{2}$'s in some $\left(A_0{}^k\right)_{ii}$, then the value of $\mathbf{p}_i$ increases indefinitely. Else, if the number of $\frac{1}{2}$'s is bigger than the number of 2^s's, the value of $\mathbf{p}_i$ will eventually become zero and stay zero from that moment on.

Thus, A_0^n is a periodic function of period k. As a consequence, the positions of all pointer variables are given by periodic functions as well, of period $k \leq 3^{\|PVar\|}$. After the execution of the stem, the variables can move at most $|\sigma|$ levels from their initial positions. Analogously, after each iteration of the loop, the difference in the level may be of at most $|\lambda|$, for each variable. Hence we need only consider trees of height at most $(\|PVar\|+1) \cdot (|\sigma| + |\lambda| \cdot k) \leq (\|PVar\|+1) \cdot \left(|\sigma| + |\lambda| \cdot 3^{\|PVar\|}\right)$. $\square$

Decidability of spuriousness is an immediate consequence of this theorem. Also, there is a bound on the number of symbolic unfoldings of a spurious lasso starting with the initial set of trees.

Corollary 1 *Let P be a program over trees, $PVar$ and $Data$ its sets of pointer variables and data elements, $\mathcal{C} = Data \times 2^{PVar}$, and $I_0 \subseteq \mathcal{T}(\mathcal{C})$ an initial set of trees. Given a lasso $S.L$ in the CA $A_{rsc}(G)$ built from an ACFG G of P, let $\sigma = \xi(S)$ and $\lambda = \xi(L)$. Then, if $\sigma.\lambda$ does not contain destructive updates, its spuriousness is decidable. Moreover, if the lasso is spurious, for all $k \geq |\lambda| \cdot \mathbf{max}(2, \|Data\|)^{2^{(\|PVar\|+1)\cdot(|\sigma|+|\lambda|\cdot 3^{\|PVar\|})}}$, we have $post(\sigma.\lambda^k, I_0) = \emptyset$.*

Proof: If χ is a composition of the $x \mapsto 2\star x$ and $x \mapsto \frac{1}{2}x$ functions, let $\#_2(\chi)$ denote the number of occurrences of 2, and $\#_{\frac{1}{2}}(\chi)$ denote the number of occurrences of $\frac{1}{2}$ in χ.

(a) To test whether $S.L$ is spurious or not, proceed as in the proof of Theorem 4, first abstracting λ into λ_a by eliminating instructions relative to the data and guards, and then build the linear system $\mathbf{p}' = A\mathbf{p} + B$ encoding the operations of λ_a, then compute A_0 from A, and iterate it to the diagonal form A_0^k. There are two cases:

1. $\#_2\left(\left(A_0^k\right)_{ii}\right) \neq \#_{\frac{1}{2}}\left(\left(A_0^k\right)_{ii}\right)$. Then either $\#_2\left(\left(A_0^k\right)_{ii}\right) > \#_{\frac{1}{2}}\left(\left(A_0^k\right)_{ii}\right)$, in which case the pointer variable corresponding to $\mathbf{p_0}_i$ will move down in the tree every k iterations, otherwise $\#_2\left(\left(A_0^k\right)_{ii}\right) < \#_{\frac{1}{2}}\left(\left(A_0^k\right)_{ii}\right)$ and it will move up. In both cases, the iteration of λ_a is bound to terminate for all input values, which implies the spuriousness of the original lasso.

the size of the matrix. The problem is to bound the maximum value of $\Pi_{i=1}^{l} m_i$ where $m_1, m_2, \ldots, m_l$ are such that $m = \sum_{i=1}^{l} m_i$. A bound of $3^{\frac{m}{3}}$ can be shown. A slight variation of this problem has been proposed at the 18^{th} International Mathematical Olympiad in 1976 [23].

2. $\#_2\left(\left(A_0^k\right)_{ii}\right) = \#_{\frac{1}{2}}\left(\left(A_0^k\right)_{ii}\right)$. Then we cannot infer termination of the lasso directly from the form of A_0^k. By Theorem 4 we have that, if the lasso is non-spurious, there is a tree of height bounded by $(\|PVar\|+1)\cdot\left(|\sigma|+|\lambda|\cdot 3^{\|PVar\|}\right)$ that witnesses non-termination. In case $\|Data\| = 1$, the number of such trees is bounded by $2^{(\|PVar\|+1)\cdot\left(|\sigma|+|\lambda|\cdot 3^{\|PVar\|}\right)}$, otherwise by $\|Data\|^{2^{(\|PVar\|+1)\cdot\left(|\sigma|+|\lambda|\cdot 3^{\|PVar\|}\right)}}$. In general, the number of trees is bounded by $\mathbf{max}(2,\|Data\|)^{2^{(\|PVar\|+1)\cdot\left(|\sigma|+|\lambda|\cdot 3^{\|PVar\|}\right)}}$.

   As the loop has no destructive updates, at most $|\lambda|\cdot\mathbf{max}(2,\|Data\|)^{2^{(\|PVar\|+1)\cdot\left(|\sigma|+|\lambda|\cdot 3^{\|PVar\|}\right)}}$ program configurations of the form $\langle l,t\rangle \in Lab\times\mathcal{T}(\mathcal{C})$ are reachable by the loop. Hence a non-terminating run along λ can be detected after at most as many steps, starting with some tree $t_{0b}\in I_0$ of bounded size.

(b) For the second part of the proof, let us suppose that $S.L$ is a spurious lasso, meaning that there is a non-terminating computation along this lasso in $A_{rsc}(G)$, but there is no such computation along $\sigma.\lambda$ in P. Then, there are again two cases:

1. $\#_2\left(\left(A_0^k\right)_{ii}\right) \neq \#_{\frac{1}{2}}\left(\left(A_0^k\right)_{ii}\right)$ for some $1\le i\le m$. Let $x\in PVar$ be the pointer variable corresponding to $\mathbf{p_0}_i$, and r_x be the counter of A, keeping track of the distance of x from the root of the tree. It can be shown, by induction of the structure of λ_a, that:

   (a) if $\#_2\left(\left(A_0^k\right)_{ii}\right) > \#_{\frac{1}{2}}\left(\left(A_0^k\right)_{ii}\right)$, then the increment of r_x in L is greater or equal to 1, and

   (b) if $\#_2\left(\left(A_0^k\right)_{ii}\right) < \#_{\frac{1}{2}}\left(\left(A_0^k\right)_{ii}\right)$, then the increment of r_x in L is smaller or equal to -1.

   In other words, the fact that a pointer is moving up or down in the tree is witnessed in both encodings, the one from the proof of Theorem 4 and the one from Table 1. As a consequence $S.L$ will terminate in $A_{rsc}(G)$ since there will be at least one r_x counter whose value will increase or decrease indefinitely. But this contradicts the hypothesis that the lasso is spurious.

2. $\#_2\left(\left(A_0^k\right)_{ii}\right) = \#_{\frac{1}{2}}\left(\left(A_0^k\right)_{ii}\right)$. In this case, all pointer variables are confined to move up or down at most $|\sigma|+|\lambda|\cdot 3^{\|PVar\|}$ levels from the initial position. Therefore, the sizes of all subtrees that are visited during any run from some initial tree $t_0\in I_0$ sum up to at most $2^{|\sigma|+|\lambda|\cdot 3^{\|PVar\|}}$. Then, the maximum number of steps taken by any finite run starting with t_0 is bounded by $|\lambda|\cdot\mathbf{max}(2,\|Data\|)^{2^{(\|PVar\|+1)\cdot\left(|\sigma|+|\lambda|\cdot 3^{\|PVar\|}\right)}}$, which is the maximum number of program configurations that can be visited at most once. Note that this is the maximum run on a spurious lasso, because any run longer than this must visit the same configuration twice, being in fact a non-terminating run. □

Despite the double exponential bound, experimental evidence (see Section 9) shows that the number of unfoldings necessary to eliminate a spurious lasso is fairly small.

7.2 Checking Spuriousness of Lassos with Destructive Updates

We now explore the case when we are given a lasso in a CA $A = \langle X, Q, q_0, \varphi_0, \rightarrow\rangle$ built from an ACFG $G = \langle Instr, L, \alpha, l_0, \longmapsto\rangle$ of a program P, and the lasso contains counter operations corresponding to rotations. For this case, we do not give a general guarantee for discovering spuriousness of lassos in a finite number of steps. However, we can give such a guarantee at least for the case when we are dealing with a program for which there exists some (albeit unknown) CA A_u simulating the program wrt. θ_{rsc} and M_{rsc}. That is, if there exists a termination argument for the program based on the tree measures we use, then we can prove spuriousness of the lasso by a symbolic iteration of the initial set. Note that in such a case, we still cannot guarantee we will find A_u nor some other CA showing termination of P (cf. Theorem 1), but we can at least get rid of particular spurious counterexamples and do not loop on the spot in our computation.

Theorem 5 *Let P be a program with an ACFG G and let $S.L$ be a spurious lasso in $A_{rsc}(G)$. If there exists a CA A_u that simulates P wrt. θ_{rsc} and M_{rsc} and that terminates on all inputs, then there exists $k \in \mathbb{N}$ such that $post(\xi(S).\xi(L)^k, I_0) = \emptyset$.*

Proof: Let $S = q_0^s \xrightarrow{\varphi_1^s} q_1^s \ldots q_{m-1}^s \xrightarrow{\varphi_m^s} q_1^l$, $L = q_1^l \xrightarrow{\varphi_1^l} q_2^l \ldots q_n^l \xrightarrow{\varphi_n^l} q_1^l$, $\sigma = \xi(S)$, and $\lambda = \xi(L)$.

We prove the theorem by contradiction. Suppose that for any $k \in \mathbb{N}$, $\sigma.\lambda^k$ is fireable for some $t \in I_0$, i.e., $post(\sigma.\lambda^k, I_0) \neq \emptyset$. We know that $\langle l_1, I_1\rangle \xrightarrow[A]{\varphi} \langle l_2, I_2\rangle$ iff $l_1 \xrightarrow[P]{\xi(\langle l_1,I_1\rangle,\varphi,\langle l_2,I_2\rangle)} l_2$, and so $\sigma.\lambda^k$ is a path in the CFG of P. Moreover, as I_0 is the set of initial trees for P, we have that $\sigma.\lambda^k$ is a run of P for any $k \in \mathbb{N}$.

Next, as the CA A_u simulates P wrt. θ_{rsc}, for any $k \in \mathbb{N}$, there must be a path $\theta_{rsc}(\sigma).\theta_{rsc}(\lambda)^k = \varphi_1^s \ldots \varphi_m^s (\varphi_1^l \ldots \varphi_n^l)^k$ in the control graph of A_u over which the simulation of $\sigma.\lambda^k$ may be run. As the set of paths $\{\theta_{rsc}(\sigma).\theta_{rsc}(\lambda)^k \mid k \geq 0\}$ is infinite and the control of A_u is finite, the control graph of A_u must contain one (or several) lassos with a stem $\theta_{rsc}(\sigma') = \theta_{rsc}(\sigma.\lambda^{k_1})$ and a loop $\theta_{rsc}(\lambda') = \theta_{rsc}(\lambda^{k_2})$ (for $k_1 \geq 0, k_2 \geq 1$)—note that $\theta_{rsc}(\lambda)$ may be partly unfolded in such lassos. The existence of a lasso of the form $\theta_{rsc}(\sigma').\theta_{rsc}(\lambda')$ in A_u then implies that there is an infinite path $\theta_{rsc}(\sigma).\theta_{rsc}(\lambda)^\omega = \varphi_1^s \ldots \varphi_m^s (\varphi_1^l \ldots \varphi_m^l)^\omega$ in A_u.

Further, the fact that $\theta_{rsc}(\sigma).\theta_{rsc}(\lambda)$ is a spurious lasso over A implies that there is a valuation $\nu_0 : X \rightarrow \mathbb{Z}$ such that $\varphi_1^s \ldots \varphi_m^s (\varphi_1^l \ldots \varphi_m^l)^\omega$ is fireable in A from ν_0. Moreover, as the initial constraint φ_0 of A is such that a valuation ν_0' satisfies φ_0 iff there exists $t' \in I_0$ such that $\forall x \in X : M_{rsc}(x, t') = \nu_0'(x)$, the initial constraint φ_0^x of A_u must be satisfiable by (at least) the same valuations as φ_0 in order to satisfy requirement (1) of Definition 1. Hence, $\nu_0 \models \varphi_0^x$ and $\varphi_1^s \ldots \varphi_m^s (\varphi_1^l \ldots \varphi_m^l)^\omega$ is fireable in A_u from ν_0 also. However, this is a contradiction as we know that A_u terminates. □

Theorem 5 implies that if we execute the program along the lasso starting with all the input trees, we will reach the empty set in a finite number of steps (corresponding to the fact that the computation cannot go on beyond some constant number of steps for any of the input trees regardless of their size). In the innermost `while` loop in the procedure presented in Section 4, this computation is interleaved with checking whether the lasso is a real one. Under the assumptions of Theorem 5, we now know that the loop terminates.

8 Abstraction Refinement

Exclusion of Spurious Lassos. If we find a spurious lasso $S.L$ in a counter automaton $A_{rsc}(G) = \langle X, Q, q_0, \varphi_0, \rightarrow\rangle$ built from an ACFG $G = \langle Instr, LI, \langle l_0, I_0\rangle, \longmapsto\rangle$ of a program P, our goal is to refine the ACFG G to $G_{S.L}$ such that the CA $A_{rsc}(G_{S.L})$ will not contain the lasso any more. For this, as we explain below, we can exploit the sets of trees generated within the symbolic execution used to show spuriousness of the lasso.

Let us have a spurious lasso with a stem $S = q_0^s \xrightarrow{\varphi_1^s} q_1^s \dots q_{m-1}^s \xrightarrow{\varphi_m^s} q_1^l$ and a loop $L = q_1^l \xrightarrow{\varphi_1^l} q_2^l \dots q_n^l \xrightarrow{\varphi_n^l} q_1^l$ in the CA $A_{rsc}(G)$. Due to the construction of $A_{rsc}(G)$, we can unambiguously map this lasso back to a lasso $q_0^s \xmapsto{i_1^s} q_1^s \dots q_{m-1}^s \xmapsto{i_m^s} q_1^l \xmapsto{i_1^l} q_2^l \dots q_n^l \xmapsto{i_n^l} q_1^l$ in G where $i_1^s = \xi(q_0, \varphi_1^s, q_1)$, ..., $i_m^s = \xi(q_{m-1}, \varphi_m^s, q_1^l)$, $i_1^l = \xi(q_1^l, \varphi_1^l, q_2^l)$, ..., $i_n^l = \xi(q_n^l, \varphi_n^l, q_1^l) \in Instr$ and $q_0^s = \langle l_0^s, I_0^s\rangle$, ..., $q_{m-1}^s = \langle l_{m-1}^s, I_{m-1}^s\rangle$, $q_1^l = \langle l_1^l, I_1^l\rangle$, ..., $q_n^l = \langle l_n^l, I_n^l\rangle \in LI$ ($l_0^s = l_0$ and $I_0^s = I_0$).

We suppose that the lasso $S.L$ was detected to be spurious via the methods described in Section 7.1 or 7.2. Hence we know that there is a $k \in \mathbb{N}$ such that the sequence of instructions $\sigma.\lambda^k$ where $\sigma = i_1^s ... i_m^s$ and $\lambda = i_1^l ... i_m^l$ is not fireable for any input tree $t \in I_0$. Let $I_0, I_1^s, I_2^s, ..., I_{m-1}^s, I_{11}^l, I_{21}^l, \dots, I_{n1}^l, I_{12}^l, I_{22}^l, ..., I_{1k}^l, I_{2k}^l, ..., I_{nk}^l, \emptyset$ be the sequence of sets of configurations that one obtains when symbolically executing the sequence of instructions $\sigma.\lambda^k$ starting with I_0 and using the implementation of the instructions over tree automata that we have.

We can create the new ACFG $G_{S.L}$ from G by splitting the sets of configurations that appear in the locations of G corresponding to the lines through which the lasso $L.S$ goes (i.e., $l_0^s, ..., l_{m-1}^s, l_1^l, ..., l_n^l$), preserving all other locations, and recomputing the transitions according to the definition of ACFG. Note that we split all locations that correspond to the program lines that appear in the lasso $S.L$, not only the locations $q_0^s, ..., q_{m-1}^s, q_1^l, ..., q_n^l$ that appear directly in $S.L$. This allows us to get rid of all copies of the same lasso that could appear in G. The splitting of the locations is done in the following two steps starting with $LI_{S.L} = LI$:

1. For i iterating from 1 to $m-1$, we replace (split) each abstract control location $\langle l_i^s, I\rangle \in LI_{S.L}$ by $\langle l_i^s, I \cap I_i^s\rangle, \langle l_i^s, I \cap \overline{I_i^s}\rangle \in LI_{S.L}$. Note that some of the control locations can be split several times because the stem can go through the appropriate line more then once.

2. For i iterating from 1 to n, we split each abstract control location $\langle l_i^l, I\rangle \in LI_{S.L}$ according to the sets of trees I_{ij}^l as follows: $\langle l_i^l, I\rangle$ will be replaced by $k+1$ new abstract control locations $\langle l_i^l, I \cap [I_{i0}^l]\rangle, \langle l_i^l, I \cap [I_{i1}^l]\rangle, ..., \langle l_i^l, I \cap [I_{ik}^l]\rangle \in LI_{S.L}$ where

   - $[I_{i0}^l] = \overline{(I_{i1}^l \cup I_{i2}^l \cup ... \cup I_{ik}^l)}$,
   - $[I_{ij}^l] = I_{ij}^l \cap \overline{(I_{i(j+1)}^l \cup ... \cup I_{ik}^l)}$ for $1 \leq j < k$ (intuitively, these sets contain trees reachable at line i in the j-th iteration of the loop λ of the lasso), and
   - $[I_{ik}^l] = I_{ik}^l$.

   Note that $\forall p \neq q : [I_{ip}^l] \cap [I_{iq}^l] = \emptyset \wedge \bigcup_{j=0}^{j \leq k} [I_{ij}^l] = I$. Note also that some of the abstract control locations can be split several times because the loop can go through the appropriate line more then once.

Theorem 6 *Suppose we have a spurious lasso with a stem S labelled with counter operations $\varphi_1^s \dots \varphi_m^s$ and a loop L labelled with counter operations $\varphi_1^l \dots \varphi_n^l$ in a CA $A_{rsc}(G)$ built from an ACFG G of a program P. Then, the CA $A_{rsc}(G_{S.L})$ built from the refined ACFG $G_{S.L}$ simulates P in the sense of Definition 1 wrt. θ_{rsc} and M_{rsc}, and it does not contain any lasso-shaped path with a stem labelled with $\varphi_1^s \dots \varphi_m^s (\varphi_1^l \dots \varphi_n^l)^p$ and a loop labelled with $(\varphi_1^l \dots \varphi_n^l)^q$ for any $p, q \geq 0$.*

Proof: The first claim follows immediately from the fact the we only split the sets of program configurations included in the particular abstract control locations (we do not loose any of the configurations), and so we can apply Lemma 1 and Theorem 2.

As for the second claim, we show below that $G_{S.L}$ does not contain a lasso-shaped path with a stem $\sigma.\lambda^p$ and a loop λ for any $p \geq 0$. This is enough to prove that in $A_{rsc}(G_{S.L})$ there is no lasso with a stem $\varphi_1^s \dots \varphi_m^s (\varphi_1^l \dots \varphi_n^l)^p$ and a loop $\varphi_1^l \dots \varphi_n^l$ for any $p \geq 0$—otherwise, from some line of the program there would have to lead two different instructions (that must come from the same program statement) with the same counter operation assigned, which is not possible in our framework.

It remains to show that $G_{S.L}$ does really not contain a lasso-shaped path with a stem $\sigma.\lambda^p$ and a loop λ for any $p \geq 0$. First, by induction on the length of the stem, it is easy to show that for any $\langle l, I \rangle \in LI_{S.L}$ reachable via a path labelled with $i_1^s \dots i_{m-1}^s$ from $\langle l_0, I_0 \rangle$, $\langle l, I \rangle \subseteq I_{m-1}^s$. Moreover, $post(i_m^s, I_{m-1}^s) = I_{11}^l$, which implies that all paths labelled with σ in $G_{S.L}$ lead to $\langle l_1^l, I' \rangle$ where $I' \subseteq [I_{1i}^l]$ for some $i \in \{1, \dots, k\}$.

Next, for any $u \in \{1, \dots, n\}$, $v \in \{1, \dots, k\}$, and any $t \in [I_{uv}^l]$, we know that $t \notin I_{uq}$ for any $q > v$ and either (1) $post(i_u^l, t)$ is not defined, (2) $u < n \ \wedge \ post(i_u^l, t) \in I_{(u+1)v}^l$, or (3) $u = n \ \wedge \ post(i_u^l, t) \in I_{1(v+1)}^l$. Moreover, from the definition of $[I_{xy}^l]$, for $u < n$, we have $I_{(u+1)v}^l \cap \bigcup_{q=0}^{q<v} [I_{(u+1)q}^l] = \emptyset$ and $I_{(u+1)v}^l \subseteq \bigcup_{q=v}^{q \leq k} [I_{(u+1)q}^l]$. Similarly, if $u = n$, we have $I_{1(v+1)}^l \cap \bigcup_{q=0}^{q<v+1} [I_{1q}^l] = \emptyset$ and $I_{1(v+1)}^l \subseteq \bigcup_{q=v+1}^{q \leq k} [I_{1q'}^l]$. Together, we then get that either (1) $post(i_u^l, t)$ is not defined, (2) $u < n \ \wedge \ post(i_u^l, t) \in [I_{(u+1)q}^l]$ for some $q \in \{v, \dots, k\}$, or (3) $u = n \ \wedge \ post(i_u^l, t) \in [I_{1q}^l]$ for some $q \in \{v+1, \dots, k\}$.

However, the above means that for each transition $\langle l, I \rangle \xrightarrow[G_{S.L}]{i} \langle l', I' \rangle$, we have $I \subseteq [I_{uv}^l]$ and $I' \subseteq [I_{u'v'}^l]$ such that $(u' > u \ \vee \ (u' = 1 \ \wedge \ v' > v))$. Consequently, as the number of the sets $[I_{xy}^l]$ is finite, the loop λ of the lasso must be broken in $G_{S.L}$. □

Beyond the Exclusion of Single Spurious Lassos. One of the reasons why the above exclusion of lassos may not be sufficient by itself is that the concrete number of transitions in a spurious lasso may restrict the sets of trees reachable along the lasso to trees of a certain form and with *some measure on the tree having a certain concrete value x*. The lasso is then excluded for these trees, but subsequently a new, slightly longer one is encountered leading to splitting wrt. trees with the measure being $x+1$, then $x+2$, etc. For example, imagine that within the stem of a lasso we go through a loop `while` $(x.\texttt{left} \neq \texttt{null})$ $x = x.\texttt{left}$ (we want to move x to the left most node) and we in particular fire the statement $x = x.\texttt{left}$ twice and then leave the `while` loop and go on in the stem. This restricts the analysis of the rest of the lasso to trees with the left-most branch of length 3, and all the splitting in the rest of the lasso will be with respect to such trees only. In the next lasso, the statement $x = x.\mathtt{left}$ can be fired 3 times, 4 times, etc. If we need to split wrt. trees of some form but with the given measure being unrestricted, we will never converge—note that a similar problem happens, e.g., also in predicate abstraction [24].

A heuristic solution we propose to the above problem is to *accelerate the splitting* in a suitable way. As shown in Section 9, we in particular successfully experimented with an *acceleration based on abstracting the sets* $[I^l_{ij}]$ used in the splitting via the finite-height abstraction from ARTMC (whose height may be iteratively adjusted in the verification loop). When using the abstraction, we split wrt. the original sets (and thus guarantee exclusion of the given spurious lasso), but also wrt. their abstraction, which according to our experience significantly increases chances of the technique to terminate (a proper theoretical examination of guarantees that can be provided for such a case is an interesting future work).

Another situation when the exclusion of spurious lassos will not itself suffice happens when the invariants computed by ARTMC are not precise enough. That is why in our verification loop, we have a certain pre-defined number that says how many times one tries to refine ACFGs and CA by splitting. If it is exhausted, one *re-runs ARTMC with a refined abstraction*—for this, we suggest to use the finite-height abstraction (possibly combined with the predicate-based one) with the abstraction height being gradually increased. Moreover, ARTMC is to be re-run over the the refined CFG of the last obtained ACFG so that the effect of the previous iterations of splitting is preserved, and the new sets of configurations computed by ARTMC are restricted to be subsets of those present in the abstract control locations of the last obtained ACFG.

9 Implementation and Experimental Results

To demonstrate the applicability of our approach, we tested its prototype implementation on several real procedures manipulating trees. In order to generate the needed line invariants and to check safety properties of the considered programs, we modified the ARTMC tool from [9]. We restricted this tool to only binary trees with parent pointers and we replaced general purpose destructive updates supported by the tool by a special implementation of tree rotations. The procedures were fully automatically verified for absence of null/undefined pointer dereferences. In some cases, we also added testers [9] to test some further safety properties. Using the generated invariants, we built ACFGs of the considered programs and the counter automata based on them. Then, we applied the ARMC tool [31] to prove termination of these counter automata (and hence termination of the programs).

We considered (1) a non-recursive *depth-first tree traversal* (for details see [22]), (2) a procedure for *searching a data value in a red-black tree* [18] (with the actual data abstracted away and all the comparisons done in a random way), and (3) the procedure that *rebalances red-black trees after inserting a new element* using tree rotations [18]. In the latter two cases, the set of input trees was a regular overapproximation of all red-black trees (we abstracted away the balancedness condition).

For these three examples, termination was shown without a need of refinement. Additionally, within the third experiment, we did two sub-experiments. To obtain the line invariants overap-

```
bool odd = true;
if (list != null) then
   while (true) do
      x = list;
      while (x != null) do
         x.data = odd;
         odd = not(odd);
         x = x.next;
      if (not(odd)) then break;
```

Figure 11: A procedure marking elements of a list as odd or even from the end of the list

proximating the sets of configurations reachable at each line, we first run ARTMC while checking both the basic memory consistency as well as shape invariance, and then we run ARTMC while checking only the basic memory consistency. In the latter case, we obtained (in a much better running time) less precise line invariants, which were, however, still precise enough for the termination proof.

The results of the experiments performed on a 1.4 GHz Xeon with 1GB of memory are summarized in Table 2. The table contains the ARTMC running times ($Time_{ARTMC}$), the number of states of the largest invariant generated by ARTMC ($|Q|_{Inv}$), the time spent by the ARMC tool to show termination ($Time_{CA}$), and the number of counters (N_{cnt})[16], locations (N_{loc}), and transitions (N_{tr}) of the CA. Taking into account the prototype nature of the tools (where, e.g., the ARTMC times could be much better if we were not building on top of the tool [9] designed for handling much more general data structures), we consider the obtained times encouraging.

Table 2: Results of experimenting with the presented method

Example	$Time_{ARTMC}$	$\|Q\|_{Inv}$	$Time_{CA}$	N_{cnt}	N_{loc}	N_{tr}
Depth-first tree traversal	43s	67	10s	5	15	20
RB-search	2s	22	1s	3	8	11
RB-rebalance after insert	1m 9s	87	36s	7	44	66

We have not yet fully implemented our refinement procedure. However, to test its usefulness, we applied it semi-automatically (with no human ingenuity added) on a case study where the initially generated invariants were not sufficient to prove termination. In particular, it was a procedure for marking elements of a singly-linked list (as a special case of a tree with no parent pointers and just a single branch) as odd or even from the end of the list (Figure 9). In the example, we slightly extend the syntax of our programs (defined in Section 3) by a support of boolean program variables with an operation $\texttt{not} : \texttt{bool} \to \texttt{bool}$ and a test $\texttt{if}\ (\texttt{bool})$ with the straightforward semantics. This does not influence the expressive power of the considered

[16] We safely skipped the use of the subtree measures in our experiments.

language as instead of a boolean variable, we could use a pointer variable x with the semantics $(x == \texttt{null}) \cong \texttt{false}$ and $(x \,!= \texttt{null}) \cong \texttt{true}$.

As the procedure does not know the length of the input list and cannot use parent pointers, it guesses what the color of the first element should be, goes through the list, and if the guess was wrong, the run is repeated once again with the right initial color. We managed to verify termination of the procedure on an arbitrary input list after excluding 9 spurious lassos (in 2 cases, the refinement was accelerated by the use the finite-height abstraction on the $[I^l_{ij}]$ sets).

10 Conclusion

We addressed the problem of proving termination of a significant class of tree manipulating programs. We provide an abstraction refinement loop based on the ARTMC framework and on exploiting the existing work on checking termination of counter automata. A number of results related to the decidability of the spuriousness problem of lasso-shaped termination counterexamples were given. Our method is not guaranteed to stop (as the problem is not r.e.), but when it stops, it provides a precise answer (both in the positive and negative case). The method was experimentally tested to be successful on several interesting practical programs.

Future work includes implementation of our refinement method and extension of our framework to more complex programs (including, e.g., unbounded dynamic allocation and general destructive pointer updates). We would also like to further investigate cases in which the universal termination problem is r.e. (and hence allowing complete verification techniques).

Acknowledgement. This work was supported in part by the French Ministry of Research (ACI project Securité Informatique), the Czech Grant Agency within projects 102/07/0322 and 102/05/H050, the Czech-French Barrande project 2-06-27, and the Czech Ministry of Education under the Research Plan No. MSM 0021630528 (“Security-Oriented Research in Information Technology”).

References

[1] P.A. Abdulla, B. Jonsson, P. Mahata, and J. d’Orso. Regular Tree Model Checking. In *Proc. of CAV’02*, volume 2404 of *LNCS*. Springer, 2002. 5.1

[2] P.A. Abdulla, A. Legay, J. d’Orso, and A.Rezine. Simulation-Based Iteration of Tree Transducers. In *Proc. of TACAS’05*, volume 3440 of *LNCS*. Springer, 2005. 5.1

[3] I. Balaban, A. Pnueli, and L.D. Zuck. Shape Analysis by Predicate Abstraction. In *Proc. of VMCAI’05*, volume 3385 of *LNCS*. Springer, 2005. 1

[4] P. Baldan, A. Corradini, J. Esparza, T. Heindel, B. König, and V. Kozioura. Verifying Red-Black Trees. In *Proc. of COSMICAH’05*, Technical report RR-05-04. Queen Mary, University of London, 2005. 1

[5] S. Bardin, A. Finkel, and E. Lozes. From Pointer Systems to Counter Systems Using Shape Analysis. In *Proc. of AVIS'06*, 2006. 1

[6] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with Lists are Counter Automata. In *Proc. of CAV'06*, volume 4144 of *LNCS*. Springer, 2006. 1

[7] A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. Verifying Programs with Dynamic 1-Selector-Linked Structures in Regular Model Checking. In *Proc. of TACAS'05*, volume 3440 of *LNCS*. Springer, 2005. 1, 4

[8] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking. *ENTCS*, 149:37–48, 2006. 5, 5.1, 5.1

[9] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In *Proc. of SAS'06*, volume 4134 of *LNCS*. Springer, 2006. 1, 4, 5.1, 9

[10] A. Bouajjani and T. Touili. Extrapolating Tree Transformations. In *Proc. of CAV'02*, volume 2404 of *LNCS*. Springer, 2002. 5.1

[11] M. Bozga, R. Iosif, and Y. Lakhnech. Storeless Semantics and Alias Logic. In *Proc. of PEPM'03*. ACM Press, 2003. 1

[12] M. Bozga, R. Iosif, and Y. Lakhnech. Flat Parametric Counter Automata. In *Proc. of ICALP'06*, volume 4052 of *LNCS*. Springer, 2006. 7, 7

[13] A.R. Bradley and Z. Manna nad H.B. Sipma. The Polyranking Principle. In *Proc. of ICALP'05*, volume 3580 of *LNCS*. Springer, 2005. 1

[14] M. Češka, P. Erlebach, and T. Vojnar. Pattern-Based Verification of Programs with Extended Linear Linked Data Structures. *ENTCS*, 145:113–130, 2006. 1

[15] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications, 2005.
URL: `http://www.grappa.univ-lille3.fr/tata`. 2, 2

[16] H. Comon and Y. Jurski. Multiple Counters Automata, Safety Analysis and Presburger Arithmetic. In *Proc. of CAV'98*, volume 1427 of *LNCS*. Springer, 1998. 7, 7

[17] B. Cook, A. Podelski, and A. Rybalchenko. Abstraction Refinement for Termination. In *Proc. of SAS'05*, volume 3672 of *LNCS*. Springer, 2005. 1

[18] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990. 9

[19] J.V. Deshmukh, E.A. Emerson, and P. Gupta. Automatic Verification of Parameterized Data Structures. In *Proc. of TACAS'06*, volume 3920 of *LNCS*. Springer, 2006. 1

[20] D. Distefano, Josh Berdine, Byron Cook, and P.W. O'Hearn. Automatic Termination Proofs for Programs with Shape-shifting Heaps. In *Proc. of CAV'06*, volume 4144 of *LNCS*. Springer, 2006. 1

[21] D. Distefano, P.W. O'Hearn, and H. Yang. A Local Shape Analysis Based on Separation Logic. In *Proc. of TACAS'06*, volume 3920 of *LNCS*. Springer, 2006. 1

[22] P. Habermehl, R. Iosif, A. Rogalewicz, and T. Vojnar. Proving Termination of Tree Manipulating Programs. Technical Report TR-2007-1, Verimag, 2007. 9

[23] International mathematical olympiad 1976.
URL: `http://www.kalva.demon.co.uk/imo/imo76.html`. 15

[24] Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental Verification by Abstraction. In *Proc. of TACAS'01*, volume 2031 of *LNCS*. Springer, 2001. 8

[25] O. Lee, H. Yang, and K. Yi. Automatic Verification of Pointer Programs Using Grammar-Based Shape Analysis. In *Proc. of ESOP'05*, volume 3444 of *LNCS*. Springer, 2005. 1

[26] A. Loginov, T.W. Reps, and M. Sagiv. Automated Verification of the Deutsch-Schorr-Waite Tree-Traversal Algorithm. In *Proc. of SAS'06*, volume 4134 of *LNCS*. Springer, 2006. 1

[27] R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate Abstraction and Canonical Abstraction for Singly-Linked Lists. In *Proc. of VMCAI'05*, volume 3385 of *LNCS*. Springer, 2005. 1

[28] M.L. Minsky. *Computation—Finite and Infinite Machines*. Prentice Hall, 1967. 4

[29] A. Møller and M.I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. of PLDI'01*. ACM Press, 2001. 1

[30] J.C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of LICS'02*. IEEE CS Press, 2002. 1

[31] A. Rybalchenko. ARMC: Abstraction Refinement Model Checker.
URL: `http://www.mpi-inf.mpg.de/~rybal/armc/`. 1, 4, 9

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

[33] E. Shahar and A. Pnueli. Acceleration in Verification of Parameterized Tree Networks. Technical Report MCS02-12, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot, Israel, 2002. 5.1

[34] E. Yahav, T. Reps, M. Sagiv, and R. Wilhelm. Verifying Temporal Heap Properties Specified via Evolution Logic. In *Proc. of ESOP'03*, volume 2618 of *LNCS*. Springer, 2003. 1

[35] T. Yavuz-Kahveci and T. Bultan. Automated Verification of Concurrent Linked Lists with Counters. In *Proc. of SAS'02*, volume 2477 of *LNCS*. Springer, 2002. 1