Automata-based Termination Proofs

Radu Iosif[1] and Adam Rogalewicz[1]

VERIMAG, CNRS, 2 av. de Vignate, F-38610 Gières, e-mail:{iosif,rogalewi}@imag.fr

Abstract. This paper proposes a framework for detecting termination of programs handling infinite and complex data domains, such as pointer structures. In this framework, the user has to specify the semantics of the instructions used by the class of programs for which the analysis is intended, as well as a finite number of well-founded relations on the data domain manipulated by these programs. The termination analysis is based on a counterexample-driven abstraction refinement loop. Our tool builds an initial abstraction of the program, which is checked for existence of potential infinite runs, by testing emptiness of its intersection with a predefined Büchi automaton. If the intersection is non-empty, a lasso-shaped counterexample is found. This counterexample is checked for spuriousness by a domain-specific procedure, and in case it is found to be spurious, the abstraction is refined, again by intersection with the complement of the Büchi automaton represented by the lasso. We have instantiated the framework for programs handling tree-like data structures, which allowed us to prove automatically termination of programs such as the depth-first tree traversal, the Deutsch-Schorr-Waite tree traversal, or the linking leaves algorithm.

1 Introduction

Discovering loop invariants is crucial for proving safety properties of programs, such as, e.g. assertion checking. Recently, in e.g. [20, 1], discovering invariant relations between program states before and after each loop iteration, has been shown useful in proving liveness properties, such as termination. These relations are called *variants* in [1].

A variant is computed from an invariant by recording the initial values of the program variables (or of other state descriptors, such as lengths of lists) into ghost variables (seeding), followed by a symbolic execution of the loop starting with the seeded invariant. In this way, a relation between the symbolic seed state and the state after one iteration (or an arbitrary number of iterations) is established. This relation typically involves *numeric quantities* and is, in general, definable in quantifier-free linear arithmetic. Well-foundedness of the variant can be (in some cases) established by existing tools [12, 21], providing an effective method to check termination of the original program.

This paper proposes a new termination analysis framework, based on the following principles:

1. We consider programs working on infinite data domains $\langle D, \preceq_1, \ldots, \preceq_n \rangle$ equipped with an arbitrary number of well-founded partial orders.
2. If $\Rightarrow \subseteq D \times D$ is any transformation induced by a program statement, and $\preceq_i$ is any partial order on D i.e., $1 \leq i \leq n$, we assume that the problem $\Rightarrow \cap \preceq_i \stackrel{?}{=} \emptyset$ is decidable algorithmically.
3. An abstraction of the program is built automatically and checked for the existence of potential non-terminating execution paths. If such a path exists, then an infinite path of the form $\sigma\lambda^\omega$ (called *lasso*) is exhibited.

4. Due to the over-approximation involved in the construction of the abstraction, the lasso found may be *spurious* i.e. it may not correspond to a real execution of the program. In this case we use domain-specific procedures to detect spuriousness, and, if the lasso is found to be spurious, the abstraction is refined by eliminating it.

The framework described here needs to be instantiated for particular classes of programs, by providing the following ingredients:

- well-founded relations $\preceq_1, \ldots, \preceq_n$ on the working domain D. In principle, their choice is naturally determined by the working domain. As an example, if D is the set of terms (trees) over a finite ranked alphabet, then $\preceq_i$ can be classical well-founded orderings on terms (e.g. Recursive Path Ordering, Knuth-Bendix Ordering, etc.).
- a decision procedure for the problems $\Rightarrow \cap \preceq_i \stackrel{?}{=} \emptyset,\ 1 \leq i \leq n$, where $\Rightarrow$ is any transition relation induced by a program statement. This is typically achieved by choosing suitable symbolic representations for relations on D, which are closed under intersection, and whose emptiness problem is decidable. For instance, this is the case when both $\Rightarrow$ and $\preceq_i$ can be encoded using finite (tree) automata.
- a decision procedure for the spuriousness problem: given a lasso $\sigma\lambda^\omega$, where σ and λ are finite sequences of program statements, does there exists an initial data element $d_0 \in D$ such that the program, started with d_0, has an infinite execution along the path $\sigma\lambda^\omega$?

In cases where the working domain D is a subset of $\mathbb{Z}^k$, for some $k > 0$, i.e. the program state can be encoded by a tuple of integer values, the choice of the well-founded relations needed to prove termination typically depends on the given program, and can be, in many cases, discovered automatically by existing techniques, e.g. ranking functions [9], transition invariants [20], etc. The main reason for which we currently ask the user to provide the relations is that our technique is geared towards data domains which cannot be encoded by a finite number of integer descriptors, such as tree-structured domains, and more complex pointer structures. Well-founded relations for classical such domains (e.g. terms over a ranked alphabet) are provided in the literature. Moreover, we are not aware of efficient techniques for automatic discovery of well-founded relations on such domains, which is an interesting topic for future research.

Providing suitable representations for the well-founded relations, as well as for the program transitions enables the framework to compute an initial abstraction of the program. The initial abstraction is an automaton which has the same control states as the program, and each edge in the control flow graph of the program is covered by one or more transitions labeled with relational symbols.

The abstraction is next checked for the existence of potentially non-terminating executions. This check uses the information provided by the well-founded relations, and excludes all lassos for which there exists a strictly decreasing well-founded relation $\succ_i,\ 1 \leq i \leq n$ that holds between the entry and exit of the loop body. This step amounts to checking non-emptiness of the intersection between the abstraction and a predefined Büchi automaton. If the intersection is empty, the original program terminates, otherwise a lasso-shaped counterexample of the form $\sigma\lambda^\omega$ is exhibited.

Deciding spuriousness of lassos is also a domain-dependent problem. For integer domains, techniques exist in cases where the transition relation of the loop is a difference bound matrix (DBM) [6] or an affine transformation with the finite monoid property [15]. For general affine transformations, the problem is currently open, as [8] mentions it[1]. For tree-structured data domains, we have shown decidability of spuriousness, in cases where the loop does not modify the structure of trees [16]. More recently, in [17], we have extended the decidability result from [16] to the cases where the loop may allocate and delete leaves in the tree.

If a lasso is found to be spurious, the program model is refined by excluding the lasso from the abstraction automaton. In our framework based on Büchi Automata, this amounts to intersecting the abstraction automaton with the complement of the Büchi Automaton representing the lasso. Since a lasso is trivially a Weak Deterministic Büchi Automaton (WDBA), complementation increases the size of the automaton by at most one state, and is done in constant time. This refinement scheme can be extended to exclude entire families of spurious lassos, also described by WDBA.

We have instantiated the framework to the verification of programs handling trees and more complex data structures with a tree-like backbone, e.g. doubly-linked lists, trees with parent pointers, trees with root pointers, or trees with linked leaves. We provide two families of well-founded relations on trees, (i) a lexicographical ordering on positions of program variables and (ii) a subset relation on nodes labeled with a given data element (from a finite domain). Program statements as well as the well-founded relations are encoded using tree automata, which provide an effective method for checking emptiness of intersections between relations. For programs on trees without destructive updates, the spuriousness problem has been shown decidable in [16], whereas for programs that allocate and delete leaves in the tree, we prove decidability of the spuriousness problem in [17]. A prototype tool has been implemented on top of the ARTMC [5] invariant generator for programs with dynamic linked data structures. Experimental results include push-button termination proofs for the Deutsch-Schorr-Waite tree traversal, as well as for the Linking Leaves procedures.

Related Work. Efficient techniques have been developed in the past for proving termination of programs with integer variables [20, 7]. This remains probably the most extensively explored class of programs, concerning termination.

Recently, techniques for programs with singly-linked lists have been developed in [3, 14, 18]. These techniques rely on tracking numeric information related to the sizes of the list segments occurring within the memory configurations reached during the execution of the program. This relevant information, expressed either as counter automata in [3], or as arithmetic relations in [14], is delivered towards tools able to establish termination on numeric domains. An extension of this method to tackle programs handling trees has been given in our previous work [16]. Unlike the works on singly-linked lists from [3, 14], where refinement (of the counter model) is typically not needed, in [16] we considered a basic form of counterexample-driven refinement, based on a symbolic unfolding of the lasso, which guarantees exclusion of the lasso from the model.

Abstraction refinement for termination has been first considered in [11], where the refinement consists in discovering and adding new well-founded relations to the set of relations used by the analysis. Since techniques for the discovery of well-founded

[1] In fact the spuriousness problem for integer affine transition relations covers another open problem, that of detecting a zero in an integer linear recurrent sequence. The latter has been shown to be NP-hard in [2] but no decidability results have been found so far.

relations (based on e.g., spurious program loops) are available only for integer domains, it is not clear for the time being whether the algorithm proposed in [11] can be also applied to programs handling pointer structures.

The framework for termination analysis presented in this paper is mainly inspired by (i) the idea of extracting variance assertions from loop invariants [1], and by (ii) our previous work on proving termination for programs handling tree-like data structures [16]. However, the method here presented distinguishes itself from the body of existing work, on the following aspects.

On one hand, the framework is general and can be instantiated to any class of programs, whose semantic domains are known to have well-founded orderings. In particular, we provide well-founded orderings on a domain of trees, e.g. lexicographical orderings, that cannot be directly encoded in quantifier-free linear arithmetic, whereas all variant assertions from [1] are confined to quantifier-free linear arithmetic. Moreover, we consider examples in which the variant relations considered track the evolution of an unbounded number of data elements (tree node labels), whereas in [1] only a finite number of seed variables could be considered.

On the other hand, the use of Büchi automata to encode the termination condition provides a natural way of finding counterexamples (by checking emptiness of Büchi automata) as well as of refining the abstract model (by intersection of the model with the complement of the counterexample). Moreover, Weak Deterministic Büchi Automata can also encode infinite sets of spurious counterexamples, which can all be eliminated at once, by one intersection with the complement of the counterexample automaton. The refinement given in [16] could only eliminate one counterexample at the time, at the cost of expanding the model by unfolding the lasso a number of times exponential in the number of program variables. In our setting, the size of the refined model is theoretically bounded by the product between the size of the model and the size of the lasso. Since, in practice the lassos are found to be rather small, the blowup caused by the refinement does not appear to be critical.

Automated checking of termination of programs manipulating trees has been also considered in [19], where the Deutsch-Schorr-Waite tree traversal algorithm was proved to terminate using a manually created progress monitor, encoded in first-order logic. In our approach, this example could be verified using the common well-founded relations on trees that is, without adding case-specific information to the framework.

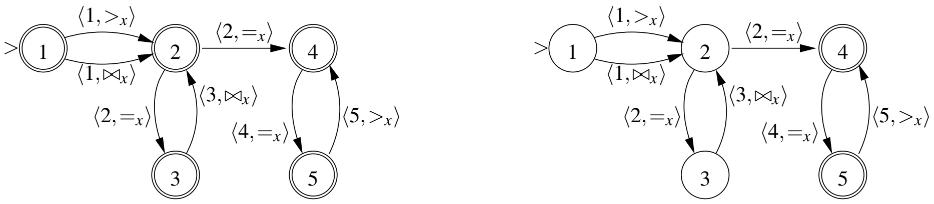
2 The Termination Analysis Framework

This section is devoted to the presentation of the termination analysis framework. We first explain the approach informally, with the aid of an example. Let us consider the program in Fig. 1 (a), working on a binary tree data structure, in which each node has two pointers to its `left`- and `right`-sons and one pointer `up` to its parent. We assume that leaves have null `left` and `right` pointers, and the `root` has a null `up` pointer.

The first loop (lines 2,3) terminates because the variable x is bound to reach a node with `x.left = null` (or `x.left.right = null`), since the tree is finite and no new nodes are created. The second loop (lines 4,5) terminates because no matter where x points to in the beginning, by going up it will eventually reach the `root` and then become null.

```
1 x := root;
2 while (x.left != null) and (x.left.right != null)
3         x := x.left.right;
4 while (x != null)
5         x := x.up;
```

(a)



(b) (c)

Fig. 1.

Let us suppose that the only well-founded ordering considered is the following: for any two trees t_1 and t_2, we have $t_1 \geq_x t_2$ if and only if the position of x in t_2 is a prefix of the position of x in t_1. Then we build the abstraction of the program given in Fig. 1 (b), where $=_x$ holds if both $\leq_x$ and $\geq_x$ hold, and $\bowtie_x$ stands for $\not\geq_x$[2].

The states in the abstract model correspond to line numbers in the original program, and every state is considered to be accepting, initially. Checking non-termination of the abstract model amounts to checking the existence of an infinite run that *does not* have a suffix of the form $=_x \ldots =_x >_x =_x \ldots =_x >_x =_x \ldots$, for otherwise, the well-foundedness of $\geq_x$ would prevent this execution from occurring in reality. Checking non-termination is done by checking emptiness of the intersection between the abstraction and the complement of the Büchi automaton recognizing the language $(\langle \_, =_x\rangle^* \langle \_, >_x\rangle)^\omega$ (cf. Fig. 2)[3]. In our case, the intersection is not empty, counterexamples being $\langle 1,>_x\rangle(\langle 2,=_x\rangle\langle 3,\bowtie_x\rangle)^\omega$ and $\langle 1,\bowtie_x\rangle(\langle 2,=_x\rangle\langle 3,\bowtie_x\rangle)^\omega$, which both correspond to the infinite execution of the first loop, i.e. lines $1(23)^\omega$.

This execution is found to be spurious by a specialized procedure that checks whether a given program lasso can be fired infinitely often. For this purpose, the method given in [16] could be used here, since the loop $1(23)^\omega$ does not change the structure of the tree. The refinement of the abstraction consists in eliminating the infinite path $1(23)^\omega$ from the model. This is done by intersecting the model with the automaton that recognizes the *complement* of the language $\{\langle 1,\geq_x\rangle,\langle 1,\bowtie_x\rangle\}(\langle 2,=_x\rangle\langle 3,\bowtie_x\rangle)^\omega$, which corresponds to the program path $1(23)^\omega$. The result of this intersection is shown Fig. 1 (c). Notice that, in this case, the refinement does not increase the size of the abstraction.

Since now, only 4 and 5 are accepting states, another intersection with the automaton in Fig. 2 will establish that the refined abstraction does not have further non-terminating executions, proving thus termination of the original program.

---

[2] Notice that we cannot assume $\geq_x$ to be a total relation on the domain of binary trees.

[3] For technical reasons that will become clear in the sequel we label the edges of the automaton with the identifier of the source states, which correspond to program lines.

2.1 Büchi Automata

This section introduces the necessary notions related to the theory of Büchi automata. Let $\Sigma = \{a, b, \ldots\}$ be a finite alphabet. We denote by Σ^* the set of finite words over Σ, and by Σ^ω we denote the set of all infinite words over Σ. For an infinite word $w \in \Sigma^\omega$, let $\inf(w)$ be the set of symbols occurring infinitely often on w. If $u, v \in \Sigma^*$ are finite words, uv^ω denotes the infinite word $uvvv\ldots$.

A *Büchi automaton* (BA) over Σ is a tuple $A = \langle S, I, \rightarrow, F \rangle$, where:

- S is a finite set of *states*,
- $I \subseteq S$ is a set of *initial states*,
- $\rightarrow \subseteq S \times \Sigma \times S$ is a *transition relation* – we denote $(s, a, s') \in \rightarrow$ by $s \xrightarrow{a} s'$,
- $F \subseteq S$ is a set of *final states*.

A run of A over an infinite word $a_0 a_1 a_2 \ldots \in \Sigma^\omega$ is an infinite sequence of states $s_0 s_1 s_2 \ldots$ such that $s_0 \in I$ and for all $i \geq 0$ we have $s_i \xrightarrow{a_i} s_{i+1}$. A run π of A is said to be *accepting* iff $\inf(\pi) \cap F \neq \emptyset$. An infinite word w is *accepted* by a Büchi automaton A iff A has an accepting run on w. The *language* of A, denoted by $\mathcal{L}(A)$, is the set of all words accepted by A.

It is well-known that Büchi-recognizable languages are closed under union, intersection and complement. For two Büchi automata A and B, let $A \otimes B$ be the automaton recognizing the language $\mathcal{L}(A) \cap \mathcal{L}(B)$. If $\|A\|$ denotes the number of states (size) of A, it can be shown that $\|A \otimes B\| \leq 3 \cdot \|A\| \cdot \|B\|$.

A Büchi automaton $A = \langle S, I, \rightarrow, F \rangle$ is said to be *complete* if for every $s \in S$ and $a \in \Sigma$ there exists $s' \in S$ such that $s \xrightarrow{a} s'$. A is said to be *deterministic* (DBA) if I is a singleton, and for each $s \in S$ and $a \in \Sigma$, there exists at most one state $s' \in S$ such that $s \xrightarrow{a} s'$. A is moreover said to be *weak* if, for each strongly connected component $C \subseteq S$, either $C \subseteq F$ or $C \cap F = \emptyset$. It is well-known that complete weak deterministic Büchi automata can be complemented by simply reverting accepting and non-accepting states. Then, for any Weak Deterministic Büchi automaton (WDBA), we have that $\|\overline{A}\| \leq \|A\| + 1$, where $\overline{A}$ is the automaton accepting the language $\Sigma^\omega \setminus \mathcal{L}(A)$—i.e. the *complement* of A.

2.2 Programs and Abstractions

In this section we introduce a model for programs handling data from a possibly infinite domain D, and define program abstractions as Büchi automata. Let $\mathfrak{I}$ be a finite set of *instructions* over a data domain $\langle D, \preceq_1, \ldots, \preceq_n \rangle$, where $\preceq_i \subseteq D \times D$ is a partial order, for $1 \leq i \leq n$. An instruction $i \in \mathfrak{I}$ is a pair $\langle g, a \rangle$ where $g \subseteq D$ is called the *guard* and $a : D \rightarrow D$ is called the *action*. An unspecified guard is assumed to be the entire domain.

A program over $\mathfrak{I}$ is a graph $P = \langle \mathfrak{I}, L, l_0, \Rightarrow \rangle$, where L is the set of *control locations*, $l_0 \in L$ is the *initial location*, and $\Rightarrow \subseteq L \times \mathfrak{I} \times L$ is the *edge relation* denoted as $l \xRightarrow{g:a} l'$. We denote by $\|P\|$ the number of control locations in P. We assume furthermore, that there is at most one instruction in between any two control locations, i.e. if $l \xRightarrow{g_1:a_1} l'$ and $l \xRightarrow{g_2:a_2} l'$ then $g_1 = g_2$ and $a_1 = a_2$. This condition is common in programming languages. In the following, we will use both textual and graphical representations of programs.

A *program configuration* is a pair $\langle l,d\rangle \in L\times D$, where l is a control location and d is a data value. An *execution* is a (possibly infinite) sequence of program configurations $\langle l_0,d_0\rangle, \langle l_1,d_1\rangle, \langle l_2,d_2\rangle, \ldots$ starting with the initial program location l_0 and some configuration $d_0 \in D$ such that, for all $i \geq 0$ there exists an edge $l_i \overset{g:a}{\Longrightarrow} l_{i+1}$ in the program, such that $d_i \in g$ and $d_{i+1} = a(d_i)$.

Let $D_0 \subseteq D$ be a set of initial data values. Then a configuration $\langle l,d\rangle$ is said to be *reachable* if there exists $d_0 \in D_0$, and the program has an execution from $\langle l_0,d_0\rangle$ to $\langle l,d\rangle$. An *invariant* of the program (with respect to the set D_0) is a function $\iota : L \to 2^D$ such that, for each $l \in L$, if $\langle l,d\rangle$ is reachable, then $d \in \iota(l)$.

Given a program $P = \langle \mathfrak{I}, L, l_0, \Rightarrow\rangle$ working over a domain $\langle D, \preceq_1, \ldots, \preceq_n\rangle$ we define the alphabet $\Sigma_{(P,D)} = L \times \{>, \bowtie, =\}^n$. For a tuple $\rho \in \{>, \bowtie, =\}^n$, we define $[\rho] \in D\times D$ as : $d\ [\langle r_1,\ldots,r_n\rangle]\ d'$ if and only if, for all $1 \leq i \leq n$:

– $d \succ_i d'$ iff r_i is $>$,
– $d \not\succeq_i d'$ iff r_i is $\bowtie$,
– $d \approx_i d'$ iff r_i is $=$.

Definition 1. *Let $P = \langle \mathfrak{I}, L, l_0, \Rightarrow\rangle$ be a program, and $\langle D, \preceq_1, \ldots, \preceq_n\rangle$ be a domain. A Büchi automaton $A = \langle S, I, \to, F\rangle$ over $\Sigma_{(P,D)}$ is said to be an abstraction of P if and only if, for every infinite execution of P : $\langle l_0,d_0\rangle\langle l_1,d_1\rangle\langle l_2,d_2\rangle\ldots$, there exists an infinite word $\langle l_0,\rho_0\rangle\langle l_1,\rho_1\rangle\langle l_2,\rho_2\rangle\ldots \in \mathcal{L}(A)$ such that $d_i\ [\rho_i]\ d_{i+1}$,for all $i \geq 0$.*

Consequently, if P has a non-terminating execution, then its abstraction A will be non-empty. However, for reasons related to the complexity of the universal termination problem, one cannot in general build an abstraction of a program that will be empty if and only if the program terminates[4].

A first question is how to build abstractions of programs effectively. We propose a method that performs under the assumption that program instructions, as well as the relations of the working domain can be symbolically represented by structures that are closed under projection, intersection and complement, and which, moreover, have a decidable emptiness problem. In the concrete case of programs with trees (see Section 3), we represent relations on trees using finite tree automata [10], who enjoy well-known closure properties.

Given a program $P = \langle \mathfrak{I}, L, l_0, \Rightarrow\rangle$ working over the domain $\langle D, \preceq_1, \ldots, \preceq_n\rangle$, and an invariant $\iota : L \to 2^D$, with respect to a set of initial data values D_0, the *initial abstraction* is the Büchi automaton $A^\iota_P = \langle L, \{l_0\}, \to, L\rangle$, where, for all $l, l' \in L$ and $\rho \in \{>, \bowtie, =\}^n$, we have :

$$l \xrightarrow{\langle l,\rho\rangle} l' \iff l \overset{g:a}{\Longrightarrow} l' \text{ and } pr_1(R_{\langle g,a\rangle} \cap [\rho]) \cap \iota(l) \neq \emptyset \tag{1}$$

where $R_{\langle g,a\rangle} = \{(d,d') \in D \mid d \in g,\ d' = a(d)\}$ and, for a relation $R \subseteq D\times D$, we denote by $pr_1(R) = \{x \mid \exists y \in D\ .\ \langle x,y\rangle \in R\}$.

Intuitively, a transition between l and l' is labeled with a tuple of relational symbols ρ if and only if there exists a program instruction between l and l' and a pair of reachable configurations $\langle l,d\rangle, \langle l',d'\rangle \in L\times D$ such that $d[\rho]d'$ and the program can move from

---

[4] The reason is that the emptiness of Büchi automata is decidable whereas the universal termination problem is not even recursively enumerable. See [16] for more details.

$\langle l,d\rangle$ to $\langle l',d'\rangle$ by executing the instruction $\langle g,a\rangle$. The intuition is that every transition relation induced by the program is "covered" by all partial orderings that have a non-empty intersection with it. Notice also that, for any $d,d' \in D$, there exists $\rho \in \{>,\bowtie,=\}^n$ such that $d\ [\rho]\ d'$. For reasons related to abstraction refinement, that will be made clear in the following, the transition in the Büchi automaton A_P^ι is also labeled with the source program location l. As an example, Fig. 1 (b) gives the initial abstraction for the program in Fig. 1 (a).

The program invariant $\iota(l)$ from (1) is needed in order to limit the coverage only to the relations involving configurations reachable at line l. In principle, we can compute a very coarse initial abstraction by considering that $\iota(l) = D$ at each program line. However, using stronger invariants enables us to compute more precise program abstractions. The following lemma proves that the initial abstraction respects Def. 1. For space reasons, all proofs have been deferred to Appendix C.

Lemma 1. *Given a program P working over the domain $\langle D,\preceq_1,\ldots,\preceq_n\rangle$, $D_0 \subseteq D$ an initial set, and $\iota : L \to 2^D$ an invariant with respect to the initial set D_0, the Büchi automaton A_P^ι is an abstraction of P.*

2.3 Checking Termination on Program Abstractions

In light of Definition 1, if a Büchi automaton A is an abstraction of a program P, then each accepting run of A reveals a *potentially* infinite execution of P. However, the set of accepting runs of a Büchi automaton is, in general not enumerable, therefore an effective termination analysis cannot attempt to check whether each run of A corresponds to a real computation of P. We propose an effective technique, based on the following:

Hypothesis 1 *The given domain is $\langle D,\preceq_1,\ldots,\preceq_n\rangle$ for a fixed $n > 0$, and the partial orders $\preceq_i$ are well-founded, for all $i = 1,\ldots,n$.*

Consequently, any infinite word $\langle l_0,\rho_0\rangle\langle l_1,\rho_1\rangle\langle l_2,\rho_2\rangle\ldots \in \mathcal{L}(A)$ from which we can extract a sequence $(\rho_0)_i(\rho_1)_i(\rho_2)_i\ldots \in (=^*>)^\omega$, for some $1 \le i \le n$, cannot correspond to a real execution of the program, in the sense of Definition 1. Therefore, we must consider only the words for which, for all $1 \le i \le n$, either:

1. there exists $K \in \mathbb{N}$ such that, $(\rho_k)_i$ is $=$, for all $k \ge K$, or
2. for infinitely many $k \in \mathbb{N}$, $(\rho_k)_i$ is $\bowtie$.

The condition above can be encoded by a Büchi automaton defined as follows. Consider that $\Sigma_{(P,D)} = L \times \{>,\bowtie,=\}^n$ is fixed. Let $S_i = \{\langle l,(r_1,\ldots,r_n)\rangle \in \Sigma_{(P,D)} \mid r_i \text{ is } \bowtie\}$ and $E_i = \{\langle l,(r_1,\ldots,r_n)\rangle \in \Sigma_{(P,D)} \mid r_i \text{ is } =\}$, for $1 \le i \le n$. With this notation, let B_i be the Büchi automaton recognizing the ω-regular language $\Sigma^*(S_i\Sigma^*)^\omega \cup \Sigma^* E_i^\omega$. This automaton is depicted in Fig. 2. Since the above condition holds for all $1 \le i \le n$, we need to compute $B = \bigotimes_{i=1}^n B_i$. Finally, the automaton which accepts all words witnessing potentially non-terminating runs of the original program is $A \otimes B$.

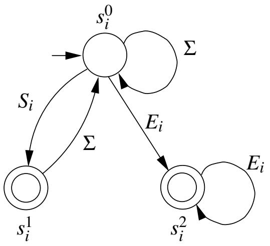


Fig. 2.

If A is an abstraction of P and $\mathcal{L}(A \otimes B) = \mathcal{L}(A) \cap \mathcal{L}(B) = \emptyset$, we can infer that P has no infinite runs. Otherwise, it is possible to exhibit a lasso-shaped non-termination witness of the form $\sigma\lambda^\omega \in \mathcal{L}(A \otimes B)$, where $\sigma, \lambda \in \Sigma^*$ are finite words labeling finite paths in $A \otimes B$. In the rest of the paper, we refer to σ as to the *stem* and to λ as to the *loop* of the lasso. The following lemma proves the existence of lasso-shaped counterexamples.

Lemma 2. *Given a well-founded domain $\langle D, \preceq_1, \ldots, \preceq_n \rangle$, A and $B = \bigotimes_{i=1}^n B_i$ Büchi automata over the alphabet $\Sigma_{(P,D)}$, if $\mathcal{L}(A \otimes B) \neq \emptyset$ then $\sigma\lambda^\omega \in \mathcal{L}(A \otimes B)$ for some $\sigma, \lambda \in \Sigma^*_{(P,D)}$, where $|\sigma|, |\lambda| \leq \|A\| \cdot (n+1) \cdot 2^n$.*

Despite the exponential bound on the size of the counterexamples, in practice it is possible to use efficient algorithms for finding lassos in Büchi automata on-the-fly, such as for instance the Nested Depth First Search algorithm [13].

2.4 Counterexample-based Abstraction Refinement

If a Büchi automaton A is an abstraction of a program $P = \langle \mathfrak{I}, L, l_0, \Rightarrow \rangle$ (cf. Def. 1), $D_0 \in D$ is a set of initial values, and $\sigma\lambda^\omega \in \mathcal{L}(A)$ is a lasso, where $\sigma = \langle l_0, \rho_0 \rangle \ldots \langle l_{|\sigma|-1}, \rho_{|\sigma|-1} \rangle$ and $\lambda = \langle l_{|\sigma|}, \rho_{|\sigma|} \rangle \ldots \langle l_{|\sigma|+|\lambda|-1}, \rho_{|\sigma|+|\lambda|-1} \rangle$, the *spuriousness problem* asks whether P has an execution along the infinite path $(l_0 \ldots l_{|\sigma|-1})(l_{|\sigma|} \ldots\ l_{|\sigma|+|\lambda|-1})^\omega$ starting with some value $d_0 \in D_0$. Notice that each pair of control locations corresponds to exactly one program instruction, therefore the sequence of instructions corresponding to the infinite unfolding of the lasso is uniquely identified by the sequences of locations $l_0, \ldots, l_{|\sigma|-1}$ and $l_{|\sigma|}, \ldots, l_{|\sigma|+|\lambda|-1}$.

Algorithms for solving the spuriousness problem exist, depending on the structure of the domain D and on the semantics of the program instructions. Details regarding spuriousness problems for integer and tree-manipulating lassos can be found in [16].

Given a lasso $\sigma\lambda^\omega \in \mathcal{L}(A)$, the refinement builds another abstraction A' of P such that $\sigma\lambda^\omega \notin \mathcal{L}(A')$. Having established that the program path $(l_0 \ldots l_{|\sigma|-1})(l_{|\sigma|} \ldots l_{|\sigma|+|\lambda|-1})^\omega$, corresponding to $\sigma\lambda^\omega$, cannot be executed for any value from the initial set, allows us to refine by excluding potentially more spurious witnesses, than just $\sigma\lambda^\omega$. Let C be the Büchi automaton recognizing the language $L_\sigma L_\lambda^\omega$, where:

$$L_\sigma = \{\langle l_0, \rho_0 \rangle \ldots \langle l_{|\sigma|-1}, \rho_{|\sigma|-1} \rangle \mid \rho_i \in \{>, \bowtie, =\}^n,\ 0 \leq i < |\sigma|\}$$
$$L_\lambda = \{\langle l_{|\sigma|}, \rho_0 \rangle \ldots \langle l_{|\sigma|+|\lambda|-1}, \rho_{|\lambda|-1} \rangle \mid \rho_i \in \{>, \bowtie, =\}^n,\ 0 \leq i < |\lambda|\}$$

Then $A' = A \otimes \overline{C}$, where $\overline{C}$ is the complement of C, is the refinement of A that excludes the lasso $\sigma\lambda^\omega$, and all other lassos corresponding to the program path $(l_0 \ldots l_{|\sigma|-1})(l_{|\sigma|} \ldots l_{|\sigma|+|\lambda|-1})^\omega$.

On the down side, complementation of Büchi automata is, in general, a costly operation: the size of the complement is bounded by $2^{O(n \log n)}$, where n is the size of the automaton, and this is also a lower bound [22]. However, the particular structure of the automata considered here comes to rescue. It can be seen that $L_\sigma L_\lambda^\omega$ can be recognized by a WDBA, hence complementation is done in constant time, and $\|A'\| \leq 3 \cdot (|\sigma| + |\lambda| + 1) \cdot \|A\|$.

Lemma 3. *Let A be a Büchi automaton that is an abstraction of a program P, and $\sigma\lambda^\omega \in \mathcal{L}(A)$ be a spurious counterexample. Then the Büchi automaton recognizing the language $\mathcal{L}(A) \setminus L_\sigma \cdot L_\lambda^\omega$ is an abstraction of P.*

This refinement technique, based on the closure of ω-regular languages under intersection and complement, could be, in principle generalized to exclude an entire family of counterexamples, described as an ω-regular language, all at once. The main difficulty here is to generalize from lasso-shaped counterexamples to more complex sets of counterexamples. In Appendix A, we provide two refinement heuristics that eliminate entire families of counterexamples, at once.

Remark In light of the fact that the universal halting problem in not r.e. (see e.g. [16] for a proof), in general, the abstraction-refinement loop is not guaranteed to terminate, even if the program terminates.

3 Proving Termination of Programs with Trees

In this section we instantiate our termination verification framework for programs manipulating tree-like data structures. We consider sequential, non-recursive C-like programs working over tree-shaped data structures with a finite set of pointer variables *PVar*. Each node in a tree contains a `data` value field, ranging over a finite set *Data* and three selector fields, denoted `left`, `right`, and `up`.[5] For $x, y \in PVar$ and $d \in Data$, we consider the programs over the set of instructions $\mathfrak{I}_T$ composed of the following :

- **guards** : $x == \mathtt{null}$, $x == y$, $x.\mathtt{data} == d$, and boolean combinations of the above,
- **actions** : $x = \mathtt{null}$, $x = y$, $x = y.\{\mathtt{left}|\mathtt{right}|\mathtt{up}\}$, $x.\mathtt{data} = d$, $x.\{\mathtt{left}|\mathtt{right}\} = \mathtt{new}$ and $x.\{\mathtt{left}|\mathtt{right}\} = \mathtt{null}$.

This set of instructions covers a large class of practical tree-manipulating procedures. For instance, Fig. 3 shows a depth-first tree traversal procedure, commonly used in real-life programs. In particular, here $PVar = \{x\}$ and $Data = \{marked, unmarked\}$. This program will be used as a running example in the rest of the section.

In order to use our framework for analyzing termination of programs with trees, we need to provide (1) well-founded partial orderings on the tree domain, (2) symbolic encodings for the partial orderings as well as for the program semantics and (3) a decision procedure for the spuriousness problem. The last point was tackled in our previous work [16], for lassos without destructive updates (i.e. instructions of the form $x.\mathtt{left}|\mathtt{right} := \mathtt{new}|\mathtt{null}$). Recently, we have developed a spuriousness detection method that works also for destructive updates of the form above[6] [17].

```
0  x := root;
1  while (x!=null)
2      if (x.left!=null) and
           (x.left.data!=mark)
3          x:=x.left;
4      else if (x.right!=null) and
           (x.right.data!=mark)
5          x:=x.right;
       else
6          x.data:=marked;
7          x:=x.up;
```

Fig. 3. Depth-first tree traversal

[5] Generalizing to trees with another arity is straightforward.

[6] Handling general updates of the form $x.\mathtt{left}|\mathtt{right} := y$ is more problematic, since in general the result of such instructions is not a tree anymore. Up to some extent, this kind of programs can be handled in the framework described in Section 3.3.

3.1 Trees and Tree Automata

For a partial mapping $f : A \rightarrow B$ we denote $f(x) = \perp$ the fact that f is undefined at some point $x \in A$. The domain of f is denoted $dom(f) = \{x \in A \mid f(x) \neq \perp\}$. For a set A we denote by $A_\perp$ the set $A \cup \{\perp\}$.

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 $\{0,1\}^*$. Let $\varepsilon \in \Pi$ denote the empty sequence, and $p.q$ denote the concatenation of sequences $p, q \in \Pi$. $p \leq_{pre} q$ denotes the fact that p is a prefix of q and $p \leq_{lex} q$ is used to denote the fact that p is less than q in the lexicographical order. We denote by $p \simeq_{pre} q$ the fact that either $p \leq_{pre} q$, or $p \geq_{pre} q$. A *tree* t over $\mathcal{C}$ is a partial mapping $t : \Pi \rightarrow \Sigma_\mathcal{C}$ 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. We denote by $\mathcal{T}(\mathcal{C})$ the set of all trees over the alphabet $\Sigma_\mathcal{C}$.

A pair of trees $(t_1, t_2) \in \mathcal{T}(\mathcal{C}_1) \times \mathcal{T}(\mathcal{C}_2)$ can be encoded by a tree over the alphabet $(\mathcal{C}_1 \cup \{\square, \perp\}) \times (\mathcal{C}_2 \cup \{\square, \perp\})$, where $\#(\langle \perp, \perp \rangle) = 0$, $\#(\langle \alpha, \perp \rangle) = \#(\langle \perp, \alpha \rangle) = \#(\alpha)$ if $\alpha \neq \perp$, and $\#(\langle \alpha_1, \alpha_2 \rangle) = \max(\#(\alpha_1), \#(\alpha_2))$. The projection functions are defined as usual i.e., for all $p \in dom(t)$ we have $pr_1(t)(p) = c_1$ if $t(p) = \langle c_1, c_2 \rangle$ and $pr_2(t)(p) = c_2$ if $t(p) = \langle c_1, c_2 \rangle$. Finally, let $\mathcal{T}(\mathcal{C}_1 \times \mathcal{C}_2) = \{t \mid pr_1(t) \in \mathcal{T}(\mathcal{C}_1) \text{ and } pr_2(t) \in \mathcal{T}(\mathcal{C}_2)\}$.

A *tree automaton* [10] over an alphabet $\Sigma_\mathcal{C}$ is a tuple $A = (Q, F, \Delta)$ 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: (i) $\square \rightarrow q$ or (ii) $c(q_1, q_2) \rightarrow q$, $c \in \mathcal{C}$.

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 $\pi(\varepsilon) \in F$. The *language* of A, denoted as $\mathcal{L}(A)$, is the set of all trees over which A has an accepting run. A set of trees $T \subseteq \mathcal{T}(\mathcal{C})$ (a tree relation $R \subseteq \mathcal{T}(\mathcal{C}_1 \times \mathcal{C}_2)$) is said to be *rational* if there exists a tree automaton A such that $\mathcal{L}(A) = T$ (respectively, $\mathcal{L}(A) = R$).

For two relations $R' \subseteq \mathcal{T}(\mathcal{C} \times \mathcal{C}')$ and $R'' \subseteq \mathcal{T}(\mathcal{C}' \times \mathcal{C}'')$ we define the composition $R' \circ R'' = \{\langle pr_1(t'), pr_2(t'') \rangle \mid t' \in R',\ t'' \in R'',\ pr_2(t') = pr_1(t'')\}$. It is well-known that rational tree languages are closed under union, intersection, complement and projection. As a consequence, rational tree relations are closed under composition.

3.2 Abstracting Programs with Trees into Büchi Automata

A memory configuration is a binary tree with nodes labeled by elements of the set $\mathcal{C} = Data \times 2^{PVar} \cup \{\square\}$ i.e., a node is either null ($\square$) or it contains a data value and a set of pointer variables pointing to it ($\langle d, V \rangle \in D \times 2^{PVar}$). Each pointer variable can point to at most one tree node (if it is null, it does not appear in the tree). For a tree $t \in \mathcal{T}(\mathcal{C})$ and a position $p \in dom(t)$ such that $t(p) = \langle d, V \rangle$, we denote $\delta_t(p) = d$ and $\nu_t(p) = V$. For clarity reasons, the semantics of the program instructions considered is given in Appendix D. First we show that all program actions considered here can be encoded as rational tree relations.

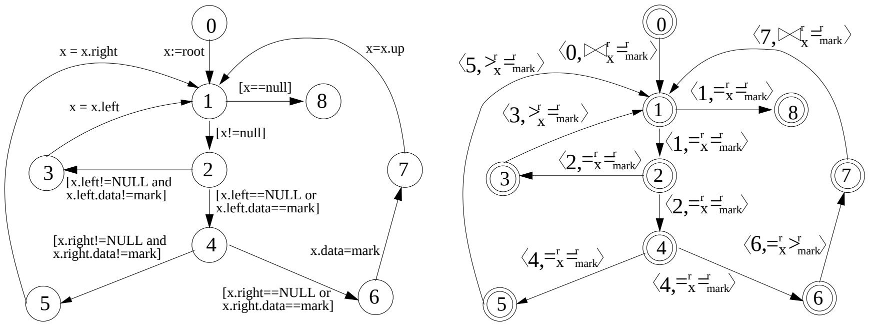


Fig. 4. The Depth-first tree traversal program and its initial abstraction

Lemma 4. *For any program instruction* $i = \langle g, a\rangle \in \mathfrak{I}_T$*, the tree relation* $R_i = \{\langle t, t'\rangle \mid t \in g,\ t' = a(t)\}$ *is rational.*

In order to abstract programs with trees as Büchi automata (cf. Definition 1), we must introduce the well-founded partial orders on the working domain. Let $D_T = \langle \mathcal{T}(\mathcal{C}), \{\preceq_x, \preceq_x^r\}_{x\in PVar}, \{\preceq_d, \preceq_d^r\}_{d\in Data}\rangle$, where:

- $t_1 \preceq_x t_2$, for some $x \in PVar$ iff there exists positions $p_1 \in dom(t_1)$, $p_2 \in dom(t_2)$ such that $x \in \nu_{t_1}(p_1)$, $x \in \nu_{t_2}(p_2)$ and $p_1 \leq_{lex} p_2$. In other words t_1 is smaller than t_2 if the position of x in t_1 is lexicographically smaller than the position of x in t_2.
- $t_1 \preceq_x^r t_2$, for some $x \in PVar$ iff (i) $dom(t_1) \subseteq dom(t_2)$, and (ii) there exists positions $p_1 \in dom(t_1)$, $p_2 \in dom(t_2)$ such that $x \in \nu_{t_1}(p_1)$, $x \in \nu_{t_2}(p_2)$ and $p_1 \geq_{lex} p_2$. In other words, t_1 is smaller than t_2 if all nodes in t_1 are also present in t_2 (i.e. no new nodes were created) and the position of x in t_1 is lexicographically bigger than the position of x in t_2.
- $t_1 \preceq_d t_2$, for some $d \in Data$ iff for any position $p \in dom(t_1)$ such that $\delta_{t_1}(p) = d$ we have $p \in dom(t_2)$ and $\delta_{t_2}(p) = d$. In other words, t_1 is smaller than t_2 if the set of nodes colored with d in t_1 is a subset of the set of nodes colored with d in t_2.
- $t_1 \preceq_d^r t_2$, for some $d \in Data$ iff (i) $dom(t_1) \subseteq dom(t_2)$, and (ii) for any position $p \in dom(t_2)$ such that $\delta_{t_2}(p) = d$ we have $p \in dom(t_1)$ and $\delta_{t_1}(p) = d$. In other words, t_1 is smaller than t_2 if all nodes in t_1 are also present in t_2 (i.e. no new nodes were created) and the set of nodes colored with d in t_2 is a subset of the set of nodes colored with d in t_1.

It can easily be shown that all relations of the form $\preceq_x$, $\preceq_x^r$, $\preceq_d$ and $\preceq_d^r$ are well-founded. Therefore the Hypothesis 1 is true for the working domain $D_T = \langle \mathcal{T}(\mathcal{C}), \{\preceq_x, \preceq_x^r\}_{x\in PVar}, \{\preceq_d, \preceq_d^r\}_{d\in Data}\rangle$, and hence the whole termination analysis framework presented in the section 2 can be employed.

The choice of relations for the time being is ad-hoc. In practice these relations are sufficient for proving termination of an important class of programs handling trees. The automatic discovery of well-founded relations falling outside the above mentioned classes can also be envisaged, however this is a subject for future work.

Lemma 5. *The relations* $\preceq_x$, $\preceq_x^r$, $x \in PVar$ *and* $\preceq_d, \preceq_d^r$, $d \in Data$ *are rational.*

The Büchi automaton representing the initial abstraction of the depth-first tree traversal procedure is depicted in Fig. 4. To simplify the figure, we use only the orders $\preceq_x^r$ and $\preceq_{mark}^r$. Thanks to these orders, there is no potential infinite run in the abstraction.

In practice, one often obtains a more precise initial abstraction by composing several program steps into one. In cases where the program instructions induce rational tree relations, it is guaranteed that composition of two or more program steps induces also a rational tree relation. In order to decide which transitions will be composed, one can use the concept of so-called cutpoints. Formally, a set of cutpoints $S \subseteq L$ is a set of control locations such that each program loop contains at least one location $l \in S$. These cutpoints can be discovered automatically as discussed in e.g., [1].

3.3 From Trees to Complex Data Structure

In general, programs with pointer data types do not manipulate just lists and trees. The structures that occur at execution time can be arbitrarily oriented graphs. In this section we use the termination detection framework for programs that manipulate such complex structures. To compute program invariants, we use the approach from [5], based on an encoding of graphs as trees with extra edges. The basic idea of this encoding is that each each structure has an underlying tree (called a *backbone*) which stays unchanged during the whole computation.

As in the previous, let *PVar* be a finite set of pointer variables, *Data* a finite set of data values, N the maximal number of selectors allowed in a single memory node, and S is a finite set of *pointer descriptors*, which are references to regular expressions (called routing expressions) over the alphabet of directions in the tree (e.g. left, right, left-up, right-up, etc.). The backbone is a tree labeled by symbols of the alphabet $\mathcal{C} = Data \times 2^{PVar} \times S^N \cup \{\square, \diamond\}$. The arity function is defined as follows: $\#(\square) = 0$ and $\#(c) = b$ for all $c \in \mathcal{C}$, where $b > 0$ is a branching factor fixed for the concrete encoding[7]. Each node of such a tree is either *active*, *removed*, or *unused*, as follows:

- *active* nodes represent memory cells present in the memory. These nodes are labeled by symbols from the alphabet $Data \times 2^{PVar} \times S^N$.
- *removed* nodes are the nodes which were deleted from the memory. They are labeled by the symbol $\diamond$. These nodes are kept in the data structure in order to preserve original shape of the underlying tree.
- *unused* nodes are labeled by $\square$ and they are placed in leaves. These nodes can be changed into active ones by a `new` statement.

Pointers between the active memory nodes are represented by pointer descriptors (placed in the active nodes). A pointer descriptor corresponding to $1 \leq i \leq N$ determines the destination of the i-th selector field of the node. There are two special designated pointer descriptors for the *null* and *undefined* pointers. The destructive updates on the data structure can be then performed simply by changing the pointer descriptors inside the tree followed by an update of the corresponding routing expression.

A set of such data structures (trees with routing expressions) can be represented by tree automata. This allows one to apply the framework of *abstract regular tree model checking* (ARTMC) [4] in order to compute (over-approximations of) the invariants of programs handling complex data structures. The ARTMC tool derives automatically the set of underlying trees and the corresponding routing expressions.

We apply the termination analysis framework along the same lines as in the previous. For a tree $t \in \mathcal{T}(\mathcal{C})$ and a position $p \in dom(t)$ such that $t(p) = \langle d, V, s_1, \ldots s_N \rangle$, we

[7] Usually, the branching factor b is equal to the number of selectors N.

denote $\delta_t(p) = d$, $\nu_t(p) = V$, and $\xi_t(p)[i] = s_i$. If $t(p) = \diamond$, $\delta_t(p) = \perp$, $\nu_t(p) = \emptyset$, and $\xi_t(p)[i] = undefined$, $1 \le i \le N$, i.e. all descriptors represent *undefined* pointers.

The working domain is $D_T = \langle \mathcal{T}(C), \{\preceq_x, \preceq_x^r\}_{x \in PVar}, \{\preceq_d, \preceq_d^r\}_{d \in Data}, \{\preceq_{1:s}, \preceq_{1:s}^r\}_{s \in S}, \ldots, \{\preceq_{N:s}, \preceq_{N:s}^r\}_{s \in S}\rangle$, where:

- $\preceq_x$, $\preceq_x^r$, $\preceq_d$ and $\preceq_d^r$ are the ones defined in Section 3.2,
- $t_1 \preceq_{i:s} t_2$, for some $1 \le i \le N$ and $s \in S$ iff for any position $p \in dom(t_1)$ such that $\xi_{t_1}(p)[i] = s$ we have $p \in dom(t_2)$ and $\xi_{t_2}(p)[i] = s$. In other words, t_1 is smaller than t_2 if the set of nodes in t_1, where the i^{th} descriptor is set to s, is a subset of the set of nodes in t_2, where the i^{th} descriptor is set to s.
- $t_1 \preceq_{i:s}^r t_2$, for some $1 \le i \le N$ and $s \in S$ iff (i) $dom(t_1) \subseteq dom(t_2)$ and (ii) for any position $p \in dom(t_2)$ such that $\xi_{t_2}(p)[i] = s$ we have $p \in dom(t_1)$ and $\xi_{t_1}(p)[i] = s$. In other words t_1 is smaller than t_2 if all nodes in t_1 are also present in t_2 (i.e. no new nodes were created) and the set of nodes in t_2, where the i^{th} descriptor is set to s, is a subset of the set of nodes in t_1, where the i^{th} descriptor is set to s.

Similar to the relations $\preceq_d$ and $\preceq_d^r$, the relation on pointer descriptor fields $\preceq_{i:s}$ and $\preceq_{i:s}^r$ are shown to be rational (hence represented by tree automata) and they are well-founded as well. As an example, Appendix B shows a procedure for linking all nodes in a tree in depth-first order. Termination of this procedure has been established using these relations.

4 Implementation and Experimental Results

We have implemented a prototype tool that uses this framework to detect termination of programs with trees and trees with routing expressions. This tool was built as an extension of the ARTMC [5] verifier for safety properties (null-pointer dereferences, memory leaks, etc.).

We applied our tool to several programs, such as manipulation procedures for doubly-linked lists, depth-first search, linking leaves in trees [5] and Deutsch-Schorr-Waite tree traversal. The results obtained on an Intel Pentium PC with 3,2GHz CPU and 1024 MB RAM memory are given in the table 1. The field *time* represents the time necessary to generate invariants and build the initial abstraction.

Example	Time
DLL-insert	3s
DLL-delete	2s
DLL-reverse	3s
Depth-first search	17s
Linking leaves in trees	18s
Deutsch-Schorr-Waite	2m 31s
Linking Nodes	9m 27s

Table 1. Results of experimenting with the prototype implementation

5 Conclusions

We proposed a new generic termination-analysis framework. In this framework, infinite runs of a program are abstracted by Büchi automata. This abstraction is then intersected with a predefined automaton representing potentially infinite runs. In case of non-empty intersection, a counterexample is exhibited. If the counterexamples is proved to be spurious, the abstraction is refined. We instantiated the framework for programs manipulating tree-like data structures and we experimented with a prototype implementation, on top of the ARTMC invariant generator. Test cases include a number of classical algorithms that manipulate tree-like data structures.

Future work includes instantiation of the method for other classes of the programs. Using the proposed method, we would like also to tackle the termination analysis for concurrent programs. Moreover, we would like to investigate methods for automated discovery of well-founded orderings on the complex data domains as trees and graphs.

References

1. J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. O'Hearn. Variance Analyses from Invariance Analyses. In *Proc. of POPL'07*. ACM Press, 2007.
2. V. D. Blondel and N. Portier. The Presence of a Zero in an Integer Linear Recurrent Sequence is NP–hard to Decide. In *Linear Algebra and Its Applications*, volume 351-352, 2002.
3. 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.
4. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking. *ENTCS*, 149:37–48, 2006. A preliminary version was presented at Infinity'05.
5. A. Bouajjani, P. Habermehl, A. Rogalewicz, T. Vojnar. Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In *Proc. of SAS'06*, *LNCS* 4134. Springer, 2006.
6. M. Bozga, R. Iosif, and Y. Lakhnech. Flat Parametric Counter Automata. In *Proc. of ICALP'06*, volume 4052 of *LNCS*. Springer, 2006.
7. A. R. Bradley, Z. Manna, and H. B. Sipma. Termination of Polynomial Programs. In *Proc. of VMCAI'2005*, volume 3385 of *LNCS*. Springer, 2005.
8. M. Braverman. Termination of Integer Linear Programs. In *Proc of CAV'06*, volume 4144 of *LNCS*. Springer, 2006.
9. M. Colón and H. Sipma. Synthesis of Linear Ranking Functions. In *Proc of TACAS'01*, volume 2031 of *LNCS*. Springer, 2001.
10. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, M. Tommasi. Tree Automata Techniques and Applications, 2005. URL: `www.grappa.univ-lille3.fr/tata`.
11. B. Cook, A. Podelski, and A. Rybalchenko. Abstraction Refinement for Termination. In *Proc. of SAS'05*, volume 3672 of *LNCS*, 2005.
12. Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond Safety. In *Proc. of CAV 2006*, volume 4144 of *LNCS*. Springer, 2006.
13. C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory Efficient Algorithms for the Verification of Temporal Properties. In *Proc. of CAV'90*, *LNCS* 531. Springer, 1991.
14. 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*, *LNCS* 4144. Springer, 2006.
15. A. Finkel and J. Leroux. How to Compose Presburger-accelerations: Applications to Broadcast Protocols. In *Proc. of FSTTCS'02*, volume 2556 of *LNCS*. Springer, 2002.
16. P. Habermehl, R. Iosif, A. Rogalewicz, and T. Vojnar. Proving Termination of Tree Manipulating Programs. In *Proc. of ATVA'07*, volume 4762 of *LNCS*. Springer, 2007.
17. R. Iosif and A. Rogalewicz. On the Spuriousness Problem for Tree Manipulating Lassos. Technical Report TR-2008-12, Verimag, 2008.
18. S.K. Lahiri and S. Qadeer. Verifying Properties of Well-Founded Linked Lists. In *Proc. of POPL'06*. ACM Press, 2006.
19. 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.
20. A. Podelski and A. Rybalchenko. Transition Invariants. In *Proc. of LICS'04*. IEEE, 2004.
21. A. Rybalchenko. The ARMC tool. URL: `http://www.mpi-inf.mpg.de/~rybal/armc/`.
22. M. Y. Vardi. The Büchi Complementation Saga. In *Proc. of STACS'07*, volume 4393 of *LNCS*. Springer, 2007.

A Refinement Heuristics

As was discussed in Section 2.4, the basic refinement technique, based on the closure of ω-regular languages under intersection and complement, could be, in principle generalized to exclude an entire family of counterexamples, described as an ω-regular language, all at once. In this appendix, we propose two refinement heuristics that eliminate entire families of counterexamples, at once. We assume in the following that we are given an invariant $\iota : L \to 2^D$ of the program.

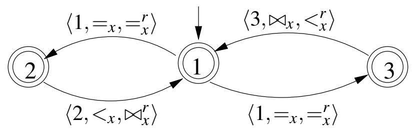
Infeasible Elementary Loop Refinement To apply this heuristic method, we suppose that there exists an upper bound $B > 0$ on the number of times λ can be iterated, starting with any data value from $\iota(l_{|\sigma|})$. The existence of such a bound can be discovered by e.g. a symbolic execution of the loop. In case such a bound exists, the language $\Sigma^*_{(P,D)} \cdot L^B_\lambda \cdot \Sigma^\omega_{(P,D)}$ is easily shown to be recognizable by a WDBA C, and the Büchi automaton $A \otimes \overline{C}$ is an abstraction of P, which excludes the spurious trace $\sigma\lambda^\omega$, as shown by the following Lemma:

Lemma 6. *Given a program $P = \langle \mathfrak{I}, L, l_0, \Rightarrow \rangle$, $\iota : L \to 2^D$ and invariant of P, A an abstraction of P, and $\lambda \in \Sigma^*_{(P,D)}$ a lasso starting and ending with $\ell \in L$. If there exists $B > 0$ such that λ^B is infeasible, for any $d \in \iota(\ell)$, then the Büchi automaton recognizing the language $\mathcal{L}(A) \setminus \Sigma^*_{(P,D)} \cdot L^B_\lambda \cdot \Sigma^\omega_{(P,D)}$ is an abstraction of P.*

As an example, let us consider the the following program:

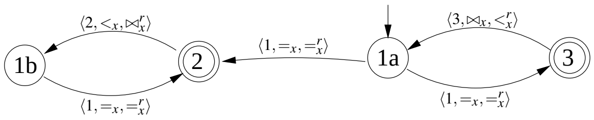
```
1 while (x != null)
        if (root.data)
2           x := x.left;
3       else x := x.up;
```

Here we assume that the root of the tree has a boolean `data` field, which is used by the loop to determine the direction (i.e., `left`, `up`) of the variable x. The initial abstraction for this program is:



This abstraction uses two well-founded relations, $\leq_x$ defined in Section 2, and $\leq^r_x$ which is a stronger version of the reversed relation $(\geq_x)$: $t_1 \leq^r_x t_2$ iff $dom(t_1) = dom(t_2)$[8] and $t_2 \leq_x t_1$.

Then a spurious lasso is $\sigma\lambda^\omega$, where σ is the empty word, and λ is $\langle 1, =_x, =^r_x \rangle \langle 2, <_x, \bowtie^r_x \rangle \langle 1, =_x, =^r_x \rangle \langle 3, \bowtie_x, <^r_x \rangle$, which corresponds to the program path 1213. This path is



infeasible for any tree, and any position of x, therefore we refine the abstraction by eliminating the language $\Sigma^*_{(P,D)} L_\lambda \Sigma^\omega_{(P,D)}$. The refined automaton is given bellow:

The intersection of the refined automaton with the automaton $B = B_1 \otimes B_2$ is empty, where $B_i,\ i = 1,2$ are the automata from Fig 2, hence, by Definition 1, we can conclude that the program terminates.

Infeasible Nested Loops Refinement Let us assume that the location $l_{|\sigma|}$ is the source (and destination) of $k > 1$ different elementary loops in A: $\lambda_1, \ldots, \lambda_k$. Moreover, let us assume that:

1. these loops can only be fired in a given total order, denoted $\lambda_{i_1} \rhd \lambda_{i_2} \rhd \ldots \lambda_{i_k}$, for each input value in the set $D_\sigma = \{d \mid \langle l_0, d_0 \rangle \stackrel{\sigma}{\Rightarrow} \langle l_{|\sigma|}, d \rangle,\ d_0 \in D_0\}$[9].
2. the infinite word $\sigma(\lambda_{i_1} \cdot \ldots \cdot \lambda_{i_k})^\omega$ is a spurious counterexample for non-termination, i.e. the corresponding program path is infeasible for any initial value from D_0.

Under these assumptions, let C be the Büchi automaton recognizing the language $L_\sigma \cdot (L_{\lambda_1} \cup \ldots \cup L_{\lambda_k})^\omega$. The following lemma shows that $A \otimes \overline{C}$ is an abstraction of the program, that excludes the spurious lasso $\sigma\lambda^\omega$:

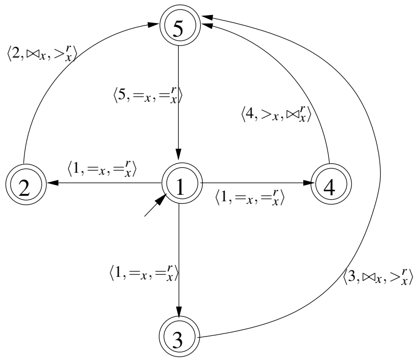
Lemma 7. *Given a program $P = \langle \mathfrak{I}, L, l_0, \Rightarrow \rangle$, $\iota : L \to 2^D$ and invariant of P, A an abstraction of P, and $\lambda_1, \ldots, \lambda_k \in \Sigma^*_{(P,D)}$ be words labeling different cycles of A starting and ending with the same location $\ell \in L$. Moreover, let $\sigma \in \Sigma^*_{(P,D)}$ be the label of a path from l_0 to ℓ in A. If there exists a total order $\lambda_{i_1} \rhd \ldots \rhd \lambda_{i_k}$ in which the cycles can be*

[8] For a tree t, $dom(t)$ is the set of positions in the tree.

[9] Notice that checking the existence of such an ordering amounts to performing at most k^2 feasibility checks, for all paths of the form $\lambda_i \cdot \lambda_j,\ 1 \le i, j \le k,\ i \ne j$.

```
1 while (x != null)
        if (root.data == 0)
2           x := x.left;
        else if (root.data == 1)
3           x := x.right;
        else if (root.data == 2)
4           x := x.up;
5       root.data := (root.data+1) % 3;
```

(a)



(b)

Fig. 5.

executed, for any $d \in \iota(\ell)$, and $\sigma \cdot (\lambda_{i_1} \cdot \ldots \cdot \lambda_{i_k})^\omega$ is moreover spurious, then the Büchi automaton recognizing the language $\mathcal{L}(A) \setminus L_\sigma \cdot (L_{\lambda_1} \cup \ldots \cup L_{\lambda_k})^\omega$ is an abstraction of P.

As an example, let us consider the program in Fig 5(a). Its initial abstraction, using the relations $\leq_x$ and $\leq_x^r$, is given in Fig. 5(b). The three nested loops are, in the unique execution order: $1251 \triangleright 1351 \triangleright 1451 \triangleright 1251$. Moreover, the composed loop $(1251)(1351)(1451)$ cannot be iterated infinitely often because the pointer variable x, moving twice down and once up in the tree, is bound to become null. This fact can be also detected automatically using e.g. the technique from [16]. Therefore we can refine the initial abstraction by eliminating the language $\{\langle 1, =_x, =_x^r \rangle \langle 2, \bowtie_x, >_x^r \rangle \langle 5, =_x, =_x^r \rangle, \langle 1, =_x, =_x^r \rangle \langle 3, \bowtie_x, >_x^r \rangle \langle 5, =_x, =_x^r \rangle, \langle 1, =_x, =_x^r \rangle \langle 4, >_x, \bowtie_x^r \rangle \langle 5, =_x, =_x^r \rangle\}^\omega$. The refined abstraction is empty, hence the program has no infinite runs.

B Example: Linking Nodes in Depth-first Order

To understand the use of the relations $\preceq_{i:s}, \preceq_{i:s}^r$ (c.f. Section 3.3), consider the program from Fig. 6. This procedure traverses a binary tree in depth-first order and links all its nodes into a cyclic singly-linked list using the selector *next*. In the beginning, all *next* selector are set to null. Then the processed nodes has the selector *next* non-null. Due to this fact, one can establish termination proof using the orders $\preceq_x^r$ and $\preceq_{next:null}$. The initial abstraction depicted in Figure 7 does not contain any potentially non-terminating run, and therefore the link_nodes procedure terminates. Note that the order based on data values (as in classical depth-first traversal) can not be used anymore to establish termination.

```
   procedure link_nodes(root)
0  x := root;
1  last:=root;
2  while (x!=null)
3      if (x.left!=null) and (x.left.next==null)
4          x:=x.left;
5      else if (x.right!=null) and (x.right.next==null)
6          x:=x.right;
       else
7          x.next:=last;
8          last:=x;
9          x:=x.up;
```

Fig. 6. linking nodes in depth-first order

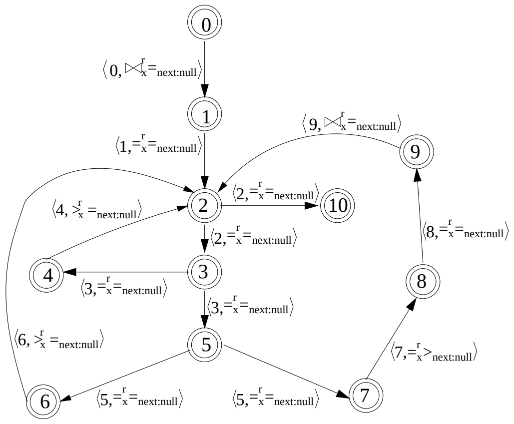


Fig. 7. Abstraction of the linking-leaves procedure

C Proofs

Proof of Lemma 1: Let $\langle l_0, d_0\rangle\langle l_1, d_1\rangle\langle l_2, d_2\rangle\ldots$ is an arbitrary infinite execution of the program P. Then for each $i \geq 0$: $l_i \stackrel{g_i:a_i}{\Longrightarrow} l_{i+1}$, $d_i \in g_i$, and $a_i(d_i) = d_{i+1}$—i.e. $(d_i, d_{i+1}) \in R_{\langle g_i, a_i\rangle}$. Moreover $d_i \in \iota(l_i)$ and $d_{i+1} \in \iota(l_{i+1})$. We know that there exists ρ_i such that $d_i[\rho_i]d_{i+1}$. Therefore for each $i \geq 0$ there exists an edge $l_i \xrightarrow{\langle l_i, \rho_i\rangle} l_{i+1}$ in the automaton A^{ι}_P i.e., the word $\langle l_0, \rho_0\rangle\langle l_1, \rho_1\rangle\langle l_2, \rho_2\rangle\ldots \in \mathcal{L}(A^{\iota}_P)$, where $d_i\ [\rho_i]\ d_{i+1}$. □

Proof of Lemma 2: If $\mathcal{L}(A \otimes B) \neq \emptyset$, then $A \otimes B$ has a run π in which at least one final state s occurs infinitely often. Let σ be the word labeling the prefix of π from the beginning to the first occurrence of s, and λ be the word labeling the segment between the first and second occurrences of s on π. Then $\sigma\lambda^{\omega} \in \mathcal{L}(A \otimes B)$.

To prove the bound on $|\lambda|$, we consider that B is the result of a generalized product $\bigotimes_{i=1}^{n} B_i$, whose states are of the form $\langle s_1, \ldots, s_n, k\rangle$ where $s_i \in \{s_i^0, s_i^1, s_i^2\}$ is a state of B_i (cf. Fig 2) and $k \in \{0, 1, \ldots, n\}$. Since λ is the label of a cycle in $A \otimes B$, the projection of the initial and final states on $B_1, \ldots, B_n$ must be the same. Then for each state in the cycle, either $s_i \in \{s_i^0, s_i^1\}$ or $s_i = s_i^2$, for each $1 \leq i \leq n$. This is because the projection of a cycle from $A \otimes B$ on B_i, $1 \leq i \leq n$ is again a cycle, and the only cycles in B_i are composed either of $\{s_i^0, s_i^1\}$ or $\{s_i^2\}$. Hence $|\lambda| \leq \|A\| \cdot (n+1) \cdot 2^n$. A similar reasoning is used to establish the bound on $|\sigma|$. □

Proof of Lemma 3 Let $\langle l_0, d_0\rangle\langle l_1, d_1\rangle\langle l_2, d_2\rangle\ldots$ be an infinite run of P. Since A is an abstraction of P, by Definition 1, there exists an infinite word $\langle l_0, \rho_0\rangle\langle l_1, \rho_1\rangle\langle l_2, \rho_2\rangle\ldots \in \mathcal{L}(A)$ such that $d_i\ [\rho_i]\ d_{i+1}$, for all $i \geq 0$. Since $\sigma\lambda^{\omega}$ is a spurious lasso, then the sequence of control locations $l_0, l_1, l_2, \ldots$ cannot correspond to the sequence of first positions from $\sigma\lambda^{\omega}$. Hence $\langle l_0, \rho_0\rangle\langle l_1, \rho_1\rangle\langle l_2, \rho_2\rangle\ldots \notin L_{\sigma} \cdot L_{\lambda}^{\omega}$. Since the infinite run $\langle l_0, d_0\rangle\langle l_1, d_1\rangle\langle l_2, d_2\rangle\ldots$ was chosen arbitrarily, it follows that the automaton recognizing $\mathcal{L}(A) \setminus L_{\sigma} \cdot L_{\lambda}^{\omega}$ is an abstraction of P. □

Proof of Lemma 4: By construction of the automata for each guard and each statement. The automaton for the guarded action is then created as composition of the corresponding guard and the action. Both the guards and actions are tree automata $A = (Q, F, \Delta)$ over the pair alphabet $\mathcal{C} \times \mathcal{C}$. We use the following subsets of the alphabet $\mathcal{C}$ within the proof:

- $N_V = \{\langle d, X\rangle \in C | \forall x \in V. x \notin X\}$
- $P_V = \{\langle d, X\rangle \in C | \forall x \in V. x \in X\}$

guards

- $x ==$ `null`: $A = (\{q_1\}, \{q_1\}, \Delta)$ with $\Delta = \{$
  - $\langle \Box, \Box\rangle \to q_1$
  - $\forall p \in N_{\{x\}} \langle p, p\rangle(q_1, q_1) \to q_1\}$
- $x == y$: $A = (\{q_1\}, \{q_1\}, \Delta)$ with $\Delta = \{$
  - $\langle \Box, \Box\rangle \to q_1$
  - $\forall p \in N_{\{x,y\}} \langle p, p\rangle(q_1, q_1) \to q_1$
  - $\forall p \in P_{\{x,y\}} \langle p, p\rangle(q_1, q_1) \to q_1\}$
- $x.$`data` $== d$: $A = (\{q_1, q_2\}, \{q_2\}, \Delta)$ with $\Delta = \{$
  - $\langle \Box, \Box\rangle \to q_1$
  - $\forall p \in N_{\{x\}}.\langle p, p\rangle(q_1, q_1) \to q_1\}$
  - $\forall \langle d, X\rangle \in P_{\{x\}}.\langle\langle d, X\rangle, \langle d, X\rangle\rangle(q_1, q_1) \to q_2$
  - $\forall p \in N_{\{x\}} \langle p, p\rangle(q_2, q_1) \to q_2$ and $\langle p, p\rangle(q_1, q_2) \to q_2\}$

actions

- $x =$ `null`: $A = (\{q_1\}, \{q_1\}, \Delta)$ with $\Delta = \{$
  - $\langle \Box, \Box\rangle \to q_1$
  - $\forall p \in N_{\{x\}} \langle p, p\rangle(q_1, q_1) \to q_1$
  - $\forall \langle d, X\rangle \in P_{\{x\}} \langle\langle d, X\rangle, \langle d, X \setminus \{x\}\rangle\rangle(q_1, q_1) \to q_1\}$
- $y = x$: $A = (\{q_1\}, \{q_1\}, \Delta)$ with $\Delta = \{$
  - $\langle \Box, \Box\rangle \to q_1$
  - $\forall p \in N_{\{x,y\}}.\langle p, p\rangle(q_1, q_1) \to q_1$
  - $\forall \langle d, X\rangle \in P_{\{x\}}.\langle\langle d, X\rangle, \langle d, X \cup \{y\}\rangle\rangle(q_1, q_1) \to q_1$
  - $\forall \langle d, X\rangle \in \{P_{\{y\}} \setminus P_{\{x\}}\}.\langle\langle d, X\rangle, \langle d, X \setminus \{y\}\rangle\rangle(q_1, q_1) \to q_1\}$
- $y = x.$`left`, $A = (\{q_1, q_y, q_2\}, \{q_2\}, \Delta)$ with $\Delta = \{$
  - $\langle \Box, \Box\rangle \to q_1$
  - $\langle \Box, \Box\rangle \to q_y$
  - $\forall \langle d, X\rangle \in N_{\{x\}}.\langle\langle d, X\rangle, \langle d, X \cup \{y\}\rangle\rangle(q_1, q_1) \to q_y$
  - $\forall \langle d, X\rangle \in N_{\{x\}}.\langle\langle d, X\rangle, \langle d, X \setminus \{y\}\rangle\rangle(q_1, q_1) \to q_1$
  - $\forall \langle d, X\rangle \in P_{\{x\}}.\langle\langle d, X\rangle, \langle d, X \setminus \{y\}\rangle\rangle(q_y, q_1) \to q_2$
  - $\forall \langle d, X\rangle \in N_{\{x\}}.\langle\langle d, X\rangle, \langle d, X \setminus \{y\}\rangle\rangle(q_2, q_1) \to q_2$
  - $\forall \langle d, X\rangle \in N_{\{x\}}.\langle\langle d, X\rangle, \langle d, X \setminus \{y\}\rangle\rangle(q_1, q_2) \to q_2\}$
- $x.$`data` $= d$: $A = (\{q_1, q_2\}, \{q_2\}, \Delta)$ with $\Delta = \{$
  - $\langle \Box, \Box\rangle \to q_1$
  - $\forall p \in N_{\{x\}}.\langle p, p\rangle(q_1, q_1) \to q_1$
  - $\forall \langle d_{orig}, X\rangle \in P_{\{x\}}.\langle\langle d_{orig}, X\rangle, \langle d, X\rangle\rangle(q_1, q_1) \to q_2$

- $\forall p \in N_{\{x\}}.\langle p,p\rangle(q_2,q_1) \rightarrow q_2$ and $\langle p,p\rangle(q_1,q_2) \rightarrow q_2\}$

– x.`left` $=$ `new`: $A = (\{q_1,q_{new},q_\perp,q_2\},\{q_2\},\Delta)$ with $\Delta = \{$
  - $\langle\Box,\Box\rangle \rightarrow q_1$
  - $\langle\perp,\Box\rangle \rightarrow q_\perp$
  - $\langle\Box,\langle d_{init},\emptyset\rangle\rangle(q_\perp,q_\perp) \rightarrow q_{new}$
  - $\forall p \in N_{\{x\}}.\langle p,p\rangle(q_1,q_1) \rightarrow q_1$
  - $\forall p \in P_{\{x\}}\langle p,p\rangle(q_{new},q_1) \rightarrow q_2$
  - $\forall p \in N_{\{x\}}.\langle p,p\rangle(q_2,q_1) \rightarrow q_2$ and $\langle p,p\rangle(q_1,q_2) \rightarrow q_2\}$

– x.`left` $=$ `null`. $A = (\{q_1,q_{del},q_\perp,q_2\},\{q_2\},\Delta)$ with $\Delta = \{$
  - $\langle\Box,\Box\rangle \rightarrow q_1$
  - $\langle\Box,\perp\rangle \rightarrow q_\perp$
  - $\forall p \in N_{\{x\}}\langle p,\Box\rangle(q_\perp,q_\perp) \rightarrow q_{del}$
  - $\forall p \in N_{\{x\}}.\langle p,p\rangle(q_1,q_1) \rightarrow q_1$
  - $\forall p \in P_{\{x\}}\langle p,p\rangle(q_{del},q_1) \rightarrow q_2$
  - $\forall p \in N_{\{x\}}.\langle p,p\rangle(q_2,q_1) \rightarrow q_2$ and $\langle p,p\rangle(q_1,q_2) \rightarrow q_2\}$

Instructions $y = x$.`right` and $y = x$.`up` are similar to the $y = x$.`left`, the instruction x.`right` $=$ `new` to the x.`left` $=$ `new`, and the instruction x.`right` $=$ `null` to the x.`left` $=$ `null`, $\Box$

Proof of lemma 5: By the construction of tree automata $A = (Q,F,\Delta)$ over the pair alphabet $(\mathcal{C} \cup \{\Box,\perp\}) \times (\mathcal{C} \cup \{\Box,\perp\})$ for each of the proposed relations.

– $t_1 \preceq_x t_2$: $A = (\{q_1,q_L,q_R,q_{acc}\},\{q_{acc}\},\Delta)$ with $\Delta = \{$
  - $\langle\Box,\Box\rangle \rightarrow q_1$
  - $\langle\perp,\Box\rangle \rightarrow q_1$
  - $\langle\Box,\perp\rangle \rightarrow q_1$
  - $\forall p_1,p_2 \in N_{\{x\}} \cup \{\Box,\perp\}.\langle p_1,p_2\rangle(q_1,q_1) \rightarrow q_1$
  - $\forall p_1,p_2 \in N_{\{x\}} \cup \{\Box,\perp\}.\langle p_1,p_2\rangle(q_L,q_1) \rightarrow q_L$
  - $\forall p_1,p_2 \in N_{\{x\}} \cup \{\Box,\perp\}.\langle p_1,p_2\rangle(q_1,q_L) \rightarrow q_L$
  - $\forall p_1,p_2 \in N_{\{x\}} \cup \{\Box,\perp\}.\langle p_1,p_2\rangle(q_R,q_1) \rightarrow q_R$
  - $\forall p_1,p_2 \in N_{\{x\}} \cup \{\Box,\perp\}.\langle p_1,p_2\rangle(q_1,q_R) \rightarrow q_R$
  - $\forall p_x \in P_{\{x\}} \forall p \in N_{\{x\}} \cup \{\Box,\perp\}.\langle p_x,p\rangle(q_1,q_1) \rightarrow q_L$
  - $\forall p_x \in P_{\{x\}} \forall p \in N_{\{x\}} \cup \{\Box,\perp\}.\langle p,p_x\rangle(q_1,q_1) \rightarrow q_R$
  - $\forall p_1,p_2 \in P_{\{x\}}.\langle p_1,p_2\rangle(q_1,q_1) \rightarrow q_{acc}$
  - $\forall p_x \in P_{\{x\}} \forall p \in N_{\{x\}}.\langle p_x,p\rangle(q_R,q_1) \rightarrow q_{acc}$
  - $\forall p_x \in P_{\{x\}} \forall p \in N_{\{x\}}.\langle p_x,p\rangle(q_1,q_R) \rightarrow q_{acc}$
  - $\forall p_1,p_2 \in N_{\{x\}}.\langle p_1,p_2\rangle(q_L,q_R) \rightarrow q_{acc}$
  - $\forall p_1,p_2 \in N_{\{x\}}.\langle p_1,p_2\rangle(q_{acc},q_1) \rightarrow q_{acc}$
  - $\forall p_1,p_2 \in N_{\{x\}}.\langle p_1,p_2\rangle(q_1,q_{acc}) \rightarrow q_{acc}$

– $t_1 \preceq_x^r t_2$: $A = (\{q_1,q_L,q_R,q_{acc}\},\{q_{acc}\},\Delta)$ with $\Delta = \{$
  - $\langle\Box,\Box\rangle \rightarrow q_1$
  - $\langle\perp,\Box\rangle \rightarrow q_1$
  - $\forall p_1,p_2 \in N_{\{x\}} \cup \{\Box,\perp\}.\langle p_1,p_2\rangle(q_1,q_1) \rightarrow q_1$
  - $\forall p_1,p_2 \in N_{\{x\}} \cup \{\Box,\perp\}.\langle p_1,p_2\rangle(q_L,q_1) \rightarrow q_L$
  - $\forall p_1,p_2 \in N_{\{x\}} \cup \{\Box,\perp\}.\langle p_1,p_2\rangle(q_1,q_L) \rightarrow q_L$
  - $\forall p_1,p_2 \in N_{\{x\}} \cup \{\Box,\perp\}.\langle p_1,p_2\rangle(q_R,q_1) \rightarrow q_R$

- $\forall p_1, p_2 \in N_{\{x\}} \cup \{\Box, \bot\}.\langle p_1, p_2\rangle(q_1, q_R) \to q_R$
- $\forall p_x \in P_{\{x\}} \forall p \in N_{\{x\}} \cup \{\Box, \bot\}.\langle p_x, p\rangle(q_1, q_1) \to q_L$
- $\forall p_x \in P_{\{x\}} \forall p \in N_{\{x\}} \cup \{\Box, \bot\}.\langle p, p_x\rangle(q_1, q_1) \to q_R$
- $\forall p_1, p_2 \in P_{\{x\}}.\langle p_1, p_2\rangle(q_1, q_1) \to q_{acc}$
- $\forall p_x \in P_{\{x\}} \forall p \in N_{\{x\}}.\langle p, p_x\rangle(q_L, q_1) \to q_{acc}$
- $\forall p_x \in P_{\{x\}} \forall p \in N_{\{x\}}.\langle p, p_x\rangle(q_1, q_L) \to q_{acc}$
- $\forall p_1, p_2 \in N_{\{x\}}.\langle p_1, p_2\rangle(q_R, q_L) \to q_{acc}$
- $\forall p_1, p_2 \in N_{\{x\}}.\langle p_1, p_2\rangle(q_{acc}, q_1) \to q_{acc}$
- $\forall p_1, p_2 \in N_{\{x\}}.\langle p_1, p_2\rangle(q_1, q_{acc}) \to q_{acc}$

– $t_1 \preceq_d t_2$: $A = (\{q_1\}, \{q_1\}, \Delta)$ with $\Delta = \{$
  - $\langle \Box, \Box\rangle \to q_1$
  - $\langle \bot, \Box\rangle \to q_1$
  - $\langle \Box, \bot\rangle \to q_1$
  - $\forall p_1, p_2 \in \mathcal{C} \cup \{\bot\}, p_1 \neq \langle d, X\rangle$ for some $X \in 2^{PVar}$ $\langle p_1, p_2\rangle(q_1, q_1) \to q_1$.
  - $\forall X_1, X_2 \in 2^{PVar}.\langle\langle d, X_1\rangle, \langle d, X_2\rangle\rangle(q_1, q_1) \to q_1\}$

– $t_1 \preceq_d^r t_2$: $A = (\{q_1\}, \{q_1\}, \Delta)$ with $\Delta = \{$
  - $\langle \Box, \Box\rangle \to q_1$
  - $\langle \bot, \Box\rangle \to q_1$
  - $\forall p_1, p_2 \in \mathcal{C} \cup \{\bot\}, p_2 \neq \langle d, X\rangle$ for some $X \in 2^{PVar}$ $\langle p_1, p_2\rangle(q_1, q_1) \to q_1$.
  - $\forall X_1, X_2 \in 2^{PVar}.\langle\langle d, X_1\rangle, \langle d, X_2\rangle\rangle(q_1, q_1) \to q_1\}$

$\Box$

Proof of Lemma 6 Let $\langle l_0, d_0\rangle\langle l_1, d_1\rangle\langle l_2, d_2\rangle \ldots$ be an infinite run of P. Since A is an abstraction of P, by Definition 1, there exists an infinite word $\langle l_0, \rho_0\rangle\langle l_1, \rho_1\rangle\langle l_2, \rho_2\rangle \ldots \in \mathcal{L}(A)$ such that $d_i\ [\rho_i]\ d_{i+1}$, for all $i \geq 0$. Since λ^B cannot be fired for any data element $d \in \iota(\ell)$, then the infinite sequence $l_0, l_1, l_2, \ldots$ may not contain the subsequence $(l_{|\sigma|} l_{|\sigma|+1} \ldots l_{|\sigma|+|\lambda|-1})^B$, corresponding to the first positions from λ^B. Consequently $\langle l_0, \rho_0\rangle\langle l_1, \rho_1\rangle\langle l_2, \rho_2\rangle \ldots \notin \Sigma^*_{(P,D)} \cdot L^B_\lambda \cdot \Sigma^\omega_{(P,D)}$. Since the infinite run $\langle l_0, d_0\rangle\langle l_1, d_1\rangle \langle l_2, d_2\rangle \ldots$ was chosen arbitrarily, it follows that the automaton recognizing $\mathcal{L}(A) \setminus \Sigma^*_{(P,D)} \cdot L^B_\lambda \cdot \Sigma^\omega_{(P,D)}$ is an abstraction of P. $\Box$

Proof of Lemma 7 Let $\langle l_0, d_0\rangle\langle l_1, d_1\rangle\langle l_2, d_2\rangle \ldots$ be an infinite run of P. Since A is an abstraction of P, by Definition 1, there exists an infinite word $\langle l_0, \rho_0\rangle\langle l_1, \rho_1\rangle\langle l_2, \rho_2\rangle \ldots \in \mathcal{L}(A)$ such that $d_i\ [\rho_i]\ d_{i+1}$, for all $i \geq 0$. We show that $\langle l_0, \rho_0\rangle\langle l_1, \rho_1\rangle\langle l_2, \rho_2\rangle \ldots \notin L_\sigma \cdot (L_{\lambda_1} \cup \ldots \cup L_{\lambda_k})^\omega$. Assume by contradiction that $\langle l_0, \rho_0\rangle\langle l_1, \rho_1\rangle\langle l_2, \rho_2\rangle \ldots \in L_\sigma \cdot (L_{\lambda_1} \cup \ldots \cup L_{\lambda_k})^\omega$. Then the infinite sequence $l_0 l_1 l_2 \ldots$ of first positions can be decomposed into a prefix corresponding to σ, followed by an infinite succession of blocks corresponding to some $\lambda_i,\ 1 \leq i \leq k$. Since $\lambda_{i_1} \rhd \lambda_{i_2} \rhd \ldots \lambda_{i_k}$ is the only possible execution order, these blocks must respect the order $\rhd$. However this is in contradiction with the fact that $\sigma(\lambda_{i_1} \cdot \ldots \cdot \lambda_{i_k})^\omega$ is a spurious lasso. The conclusion follows. $\Box$

D Concrete Semantics of Tree Manipulating Programs

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

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

$$\frac{\begin{array}{c} p \in dom(t) \qquad y \in \nu(t(p)) \\ t(p.0) = \square \qquad \langle l,t\rangle \xrightarrow{x=null} \langle l',t'\rangle \end{array}}{\langle l,t\rangle \to \langle l',t'\rangle} \qquad \frac{\forall p \in dom(t)\ .\ y \notin \nu(t(p))}{\langle l,t\rangle \to Err} \quad x = y.\texttt{left}(b)$$

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

$$\frac{\begin{array}{cc} d \in Data & p \in dom(t) \\ x \in \nu(t(p)) & t(p.0) = \square \end{array}}{\langle l,t\rangle \to \langle l', t[p.0 \leftarrow \langle d, \emptyset\rangle, p.0.0 \leftarrow \square, p.0.1 \leftarrow \square]\rangle} \qquad \frac{\begin{array}{l} (\forall p \in dom(t)\ .\ x \notin \nu(t(p))) \vee \\ (p \in dom(t)\ \wedge\ x \in \nu(t(p))\ \wedge\ t(p.0) \neq \square) \end{array}}{\langle l,t\rangle \to Err} \quad x.\texttt{left} = new$$

$$\frac{\begin{array}{c} p \in dom(t) \quad x \in \nu(t(p)) \quad p.0 \in dom(t) \\ t(p.0.0) \quad = \quad t(p.0.1) \quad = \quad \square \end{array}}{\langle l,t\rangle \to \langle l', t[p.0 \leftarrow \square, p.0.0 \leftarrow \bot, p.0.1 \leftarrow \bot]\rangle} \qquad \frac{\begin{array}{l} (\forall p \in dom(t)\ .\ x \notin \nu(t(p))) \vee \\ (p \in dom(t)\ \wedge\ x \in \nu(t(p)) \wedge \\ \quad (t(p.0) = \square \vee t(p.0.0) \neq \square\ \vee\ t(p.0.1) \neq \square)) \end{array}}{\langle l,t\rangle \to Err} \quad x.\texttt{left} = null$$

Fig. 8. The concrete semantics of program statements