Restricting Tree Grammars with Term Rewriting

Jan Bessai
TU Dortmund, Germany

Lukasz Czajka
TU Dortmund, Germany

Felix Laarmann
TU Dortmund, Germany

Jakob Rehof
TU Dortmund, Germany

Abstract

We investigate the problem of enumerating all terms generated by a tree-grammar which are also in normal form with respect to a set of directed equations (rewriting relation). To this end we show that deciding emptiness and finiteness of the resulting set is EXPTIME-complete. The emptiness result is inspired by a prior result by Comon and Jacquemard on ground reducibility. The finiteness result is based on modification of pumping arguments used by Comon and Jacquemard. We highlight practical applications and limitations. We provide and evaluate a prototype implementation. Limitations are somewhat surprising in that, while deciding emptiness and finiteness is EXPTIME-complete for linear and nonlinear rewrite relations, the linear case is practically feasible while the nonlinear case is infeasible, even for a trivially small example. The algorithms provided for the linear case also improve on prior practical results by Kallat et al.

2012 ACM Subject Classification Theory of computation → Tree languages; Theory of computation → Automata extensions; Theory of computation → Equational logic and rewriting

Keywords and phrases tree automata, tree grammar, term rewriting, normalization, emptiness, finiteness

Digital Object Identifier 10.4230/LIPIcs.FSCD.2022.14

Acknowledgements We thank Christoph Stahl for creating the `tikz` figures. We also thank our reviewers for their insightful and useful comments which improved the final version of the paper.

1 Introduction

Suppose we are given a tree grammar G over a ranked alphabet $\mathcal{F}$ and a rewriting relation R over terms generated from $\mathcal{F}$. We are interested in deciding emptiness and finiteness of the set $L(G) \cap \mathrm{NF}(R)$, where $\mathrm{NF}(R)$ is the set of terms in normal form with respect to R. This problem may arise naturally in situations where trees recognized by G are subject to simplifications under R and we are only interested in simplified terms. For example, we may think of G as recognizing a language of algebraic expressions including, say, expressions of the form $f(a, b)$, and R captures simplifications under algebraic laws, say, idempotence $f(X, X) \to X$.

Our interest in this problem arose in the context of work on component-based synthesis [18], specifically combinatory logic synthesis (CLS). CLS is based on solving bounded versions of the inhabitation problem for combinatory logic with intersection types [17, 8] and has been implemented in the CLS-framework (see [3] for a fairly recent description). CLS has been applied in a number of contexts, recent examples include [4, 21, 10, 19].

In CLS, the (possibly infinite) solution set to a synthesis query is a set of combinatory terms (each representing a program or a metaprogram), which is represented by a tree grammar G recognizing combinatory terms. Here, R acts as a filter restricting the solution set to normal forms in $L(G) \cap \mathrm{NF}(R)$, and we are interested in enumerating normal solutions. Since the filter specified by R might well lead to a finite set of normal solutions even though $L(G)$ is infinite, knowing whether $L(G) \cap \mathrm{NF}(R)$ is empty or finite is of immediate interest. The results reported in the present paper form the basis of a prototype implementation intended to become an extension to the CLS-framework.

Notice that the problems considered here are entirely different from the problem of recognizing the normal forms (wrt. R) of $L(G)$ for a given grammar of terms G. The latter problem is obviously undecidable (take G to recognize a given SKI-term, and we would need to solve the halting problem for SKI-calculus), but it is also not relevant for our purposes, since we are interested only in terms that are already contained in the solution set $L(G)$. In our setting, the rewriting relation R is used as a filter such that only the left-hand sides of rules matter to filter out non-normal forms from $L(G)$ (essentially, by solving the problem of non-matchability of terms with any left-hand side of R).

1.1 Contributions

Our contribution is twofold. First, we prove EXPTIME-completeness of emptiness and finiteness of $L(G) \cap \mathrm{NF}(R)$. Our techniques draw on previous work by Comon and Jacquemard (see Section 1.2) on automata with disequality constraints (ADC) for the ground reducibility problem. Disequality constraints are necessary to handle nonlinear rules in R. Our main technical contribution is contained in the Bound Theorem (Theorem 7), which provides a bound on the maximum height of accepted terms, when $L(G) \cap \mathrm{NF}(R)$ is finite. The bound follows from a pumping argument for finiteness and acts as an upper bound for enumeration in the finite case.

Second, we provide experimental analysis of the algorithm for deciding emptiness and finiteness provided here, based on a Haskell implementation. It turns out that, even though the left-linear restriction (wrt. R) is somewhat surprisingly already EXPTIME-complete for both problems (Theorem 23, Theorem 25), the performance in the nonlinear case is orders of magnitude worse than in the linear case. Our analysis shows that the nonlinear case reaches an order of magnitude of worst-case performance (rendering it infeasible for even trivially small examples), whereas the linear restriction can be engineered to be practically feasible, improving on a previously published algorithm. Whether one can find heuristics to engineer the nonlinear case for practically interesting cases is left as a question for future research.

1.2 Related work

The theoretical results in the present paper are adaptations and extensions of the results of Comon and Jacquemard on the EXPTIME-completeness of the ground reducibility problem [7, 6]. We consider different problems of emptiness and finiteness of the intersection of a regular tree language with the set of normal forms of a rewrite system. While the proof of our Bound Theorem and the automata constructions draw heavily from [7], the adaptation of the results to emptiness and finiteness is not trivial.

The EXPTIME-completeness of the emptiness problem was essentially shown by Comon and Jacquemard [7] (only relatively small adjustments are necessary to adapt their arguments to our problem). An EXPTIME algorithm for finiteness was essentially already obtained in [9], where [9, Lemma 5.19] corresponds to our Bound Theorem and the constructions in the proofs are similar. However, our exponential bound is better than the exponential bound given in [9], which may have practical implications. EXPTIME-hardness of the finiteness problem seems to be new.

Our results depend on the notion of automata with disequality constraints (ADCs) introduced by Comon and Jacquemard [7]. Related automata frameworks are tree automata with normalization [16] and equational tree automata [15]. In these frameworks, the automata transitions are defined modulo normalization or an equational theory resulting in accepted languages closed under these operations, while we are interested in restrictions of regular tree languages to normal forms of a rewrite system. Another related model is tree automata with global constraints [13] where the constraints are associated with pairs of states and enforce equality or disequality of all subterms at all nodes where the states appear in the corresponding run, in contrast to ADCs where the constraints are local and associated with transition rules, enforcing disequality of subterms at a given transition.

Going beyond the previously described usecase for CLS, other synthesis frameworks might profit from our approach. For example, Madhusudan [14] describes a framework for synthesizing reactive programs. This approach is similar to recent additions to the broader field of syntax guided synthesis [12]. In both cases, synthesized programs are represented by trees and constructed from tree-languages, that are then restricted to match desired program semantics. In the present paper we are not concerned with arbitrary semantic specifications, but just equations for program normalization. In synthesis frameworks such as the above, this might be a useful way to reduce the search space or filter solutions.

2 Preliminaries

In this section we fix notations and recall standard definitions related to tree grammars and term rewriting. See e.g. [5] and, respectively, [1, 20] for more thorough introductions to these topics.

By $\mathcal{T}(\mathcal{F}, X)$ we denote the set of all first-order terms over the signature $\mathcal{F}$ with variables taken from the set X. The set of *ground terms* $\mathcal{T}(\mathcal{F}, \emptyset)$ is also denoted by $\mathcal{T}(\mathcal{F})$. By ϵ we denote the empty string, by $\cdot$ the concatenation operation on strings, and by $[i]$ the string consisting of a single letter i. The set of *positions* of a term $t \in \mathcal{T}(\mathcal{F}, X)$ is a set $\mathrm{Pos}(t)$ of strings of positive integers defined by: (1) if $t = x$ then $\mathrm{Pos}(t) = \{\epsilon\}$; (2) if $t = f(t_1, \ldots, t_n)$ then $\mathrm{Pos}(t) = \{\epsilon\} \cup \bigcup_{i=1}^{n} \{[i] \cdot p \mid p \in \mathrm{Pos}(t_i)\}$. The *size* of a term t is the cardinality of $\mathrm{Pos}(t)$. The *prefix order* on positions is defined by: $p \preceq q$ iff there is p' with $p \cdot p' = q$. For $p \in \mathrm{Pos}(t)$, the *subterm* of t at position p is denoted by $t|_p$. By $t(p)$ we denote the symbol in t at position p. The *replacement* $t[s]_p$ is the term obtained from s by replacing the subterm at position p with s. By $\mathrm{Var}(t)$ we denote the set of variables occurring in t. A context C is a term in $\mathcal{T}(\mathcal{F}, X \cup \{\Box\})$ such that $\Box$ occurs in C exactly once. By $C[t]$ we denote the term in $\mathcal{T}(\mathcal{F}, X)$ obtained from C by replacing $\Box$ with t.

A *term rewriting system* (TRS) R is a set of rules $t \to s$ such that $\mathrm{Var}(s) \subseteq \mathrm{Var}(t)$ and t is not a variable. We denote by $\to_R$ the *reduction relation* associated with the TRS R: $t \to_R s$ iff there is a rule $l \to r \in R$, a context C and a substitution σ such that $t = C[\sigma l]$ and $s = C[\sigma r]$. A term t is in *normal form* if there is no t' with $t \to_R t'$. The *size* $||R||$ of the TRS R is the sum of the sizes of the left-hand sides of rules in R.

For any binary relation $\to$, by $\to^*$ we denote the transitive-reflexive, and by $\to^+$ the transitive closure of $\to$.

A regular tree grammar is a tuple $G = (S, N, \mathcal{F}, R_G)$ such that $S \in N$ is the start symbol, N is a set of nullary nonterminals, $\mathcal{F}$ is a set of terminals, R_G is a set of production rules of the form $A \to \alpha$ where $A \in N$ and $\alpha \in \mathcal{T}(\mathcal{F} \cup N)$. The *derivation relation* associated with G is defined by: $t \to_G s$ iff there is a rule $A \to \alpha \in R$ and a context C such that $t = C[A]$ and $s = C[\alpha]$. The *language generated* by G is defined by $L(G) = \{t \in \mathcal{T}(\mathcal{F}) \mid S \to_G^+ t\}$.

A finite tree automaton over signature $\mathcal{F}$ is a tuple $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ where Q is a set of states, $Q_f \subseteq Q$ is a set of final states, and Δ is a set of transition rules of the form $f(q_1, \ldots, q_n) \to q$ with $f \in \mathcal{F}_n$ (i.e. f is an n-ary symbol in $\mathcal{F}$), $q, q_1, \ldots, q_n \in Q$. The move relation $\to_{\mathcal{A}}$ is defined by: $t \to_{\mathcal{A}} t'$ iff there are a transition rule $l \to r$ and a context C with $t = C[l]$ and $t' = C[r]$. A ground term $t \in \mathcal{T}(\mathcal{F})$ is *accepted* by $\mathcal{A}$ if there is $q_f \in Q_f$ with $t \to_{\mathcal{A}}^* q_f$. The language $L(\mathcal{A})$ recognized by $\mathcal{A}$ is the set of all terms accepted by $\mathcal{A}$.

In terms of the recognized languages, finite tree automata and regular tree grammars are equivalent. A *regular tree language* is a language recognized by a finite tree automaton, or equivalently a language generated by a regular tree grammar.

3 Automata with disequality constraints

Automata with disequality constraints (ADC) were introduced by Comon and Jacquemard in [6, 7]. These are essentially tree automata where some rules may additionally check whether two subterms at given positions are not equal. The idea is to construct a normal forms ADC which recognises exactly the normal forms of a given term rewriting system. The disequality constraints are needed to handle non-left-linear rules. To check emptiness or finiteness of the intersection, a product automaton is created. The construction of the normal forms ADC has already been presented by Comon and Jacquemard. In this section, we recall the definition of ADCs and related notions. The constructions of the normal forms automaton and the finiteness checking algorithms are presented in subsequent sections.

Definitions in this section are either verbatim copies or minor modifications of those in [7].

▶ **Definition 1.** *An* automaton with disequality constraints *(ADC) is a tuple* (Q, Q^f, Δ) *where Q is a finite set of states, $Q^f \subseteq Q$ is the set of final states, and Δ is a finite set of transition rules of the form $f(q_1, \ldots, q_n) \xrightarrow{c} q$ where $f \in \mathcal{F}^n$, $q_1, \ldots, q_n, q \in Q$ and c is a Boolean combination without negation of constraints $p_1 \neq p_2$ with p_1, p_2 positions. A term $t \in \mathcal{T}(\mathcal{F})$* satisfies the constraint *$p_1 \neq p_2$, denoted $t \models p_1 \neq p_2$, if both $p_1, p_2 \in \mathrm{Pos}(t)$ and $t|_{p_1} \neq t|_{p_2}$. A* run *of an automaton $\mathcal{A} = (Q, Q^f, \Delta)$ on a term t is a term ρ over signature Δ (i.e. each rule $r = (f(q_1, \ldots, q_n) \to q) \in \Delta$ is treated as an n-ary symbol) such that for all $p \in \mathrm{Pos}(t)$, if $t(p) = f \in \mathcal{F}^n$ then $\rho(p)$ is a rule $f(q_1, \ldots, q_n) \xrightarrow{c} q$ and:*

1. *$\rho(p \cdot [i])$ is a rule with target q_i, for $i = 1, \ldots, n$ (weak),*
2. *$t|_p \models c$ (strong).*

If only the first condition (weak) is satisfied by ρ, then ρ is a weak run.

A ground term $t \in \mathcal{T}(\mathcal{F})$ is accepted *by $\mathcal{A}$ if there is a run ρ of $\mathcal{A}$ on t such that $\rho(\epsilon)$ is a rule whose target is a final state in Q^f. The* language *$L(\mathcal{A})$ of $\mathcal{A}$ is the set of terms accepted by $\mathcal{A}$.*

▶ Note 2.

- An ADC with all constraints $\top$ is a finite tree automaton (the constraints are always satisfied).
- An ADC can be non-deterministic (more than one run on some term) or not completely specified (no run on some term).
- The term used in the construction of a run ρ is denoted as the associated term$(\rho) \in \mathcal{T}(\mathcal{F})$.

▶ **Example 3.** Let $\mathcal{F} = \{f, a, b\}$ and $Q = \{q\} = Q^f$.

$$\Delta = \{r_1 : a \to q, r_2 : b \to q, r_3 : f(q, q) \xrightarrow{1 \neq 2} q\}.$$

The term $f(a,b)$ is accepted because $\rho = r_3(r_1, r_2)$ is a run on t and r_3 yields a final state. The term $f(a,a)$ is not accepted: there is a weak run $r_3(r_1, r_1)$ but the disequality of r_3 is not satisfied. In general, the automaton accepts ground terms irreducible by a TRS with a single rule with the left-hand side $f(x,x)$.

▶ **Definition 4.** *Let $\mathcal{A}$ be an ADC.*

Let $\mathcal{C}(\mathcal{A})$ be the set of all triples (β, π, π') such that β is a prefix of π' and $\pi \neq \pi'$ or $\pi' \neq \pi$ is an atom occurring in a constraint of transition rules of $\mathcal{A}$. Let $c(\mathcal{A}) = |\mathcal{C}(\mathcal{A})|$.

Let $\mathcal{S}(\mathcal{A})$ be the set of all suffixes of positions π, π' in an atom $\pi \neq \pi'$ occurring in a constraint of a rule in $\mathcal{A}$. Let $s(\mathcal{A}) = |\mathcal{S}(\mathcal{A})|$.

We define $d(\mathcal{A})$ as the maximum length of π in a constraint $\pi \neq \pi'$ or $\pi' \neq \pi$ in $\mathcal{A}$. By $n(\mathcal{A})$ we denote the maximum number of atomic constraints occurring in a rule of $\mathcal{A}$.

Note that $c(\mathcal{A}), s(\mathcal{A}) \leq |\mathcal{A}|^2$ and $d(\mathcal{A}), n(\mathcal{A}) \leq |\mathcal{A}|$ and $d(\mathcal{A}) \leq s(\mathcal{A})$. In [7], $c(\mathcal{A})$ and $\mathcal{C}(\mathcal{A})$ are used instead of $s(\mathcal{A})$ and $\mathcal{S}(\mathcal{A})$. Our definitions of $c(\mathcal{A})$ and $\mathcal{C}(\mathcal{A})$ are modifications of the definitions from [7] to upward pumping.

▶ **Definition 5.** *Let $\mathcal{A} = (Q, Q^f, \Delta)$ be an ADC and ρ a weak run of $\mathcal{A}$ on t. An* equality *of ρ is a triple of positions (p, π, π') such that $p, p \cdot \pi, p \cdot \pi' \in \mathrm{Pos}(t)$, $\pi \neq \pi'$ is in the constraint of $\rho(p)$ and $t|_{p\cdot\pi} = t|_{p\cdot\pi'}$.*

An equality (p', π, π') in a weak run ρ is classified according to a particular position $p \in \mathrm{Pos}(t)$:

- *It is* close *to p if $p' \preceq p \prec p' \cdot \pi$ or $p' \preceq p \prec p' \cdot \pi'$,*
- *It is* far *from p if $p' \cdot \pi \preceq p$ or $p' \cdot \pi' \preceq p$.*

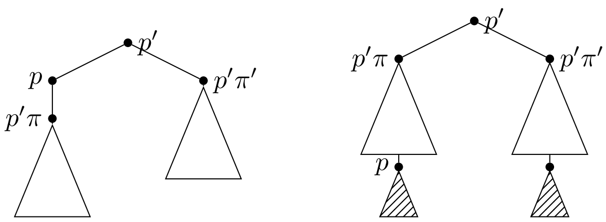

Figure 1 Equality close to p (left) and equality far from p (right).

▶ **Lemma 6.** *Every equality in $\rho[\rho']_p$ is either far from p or close to p.*

Proof. Identical to the proof of Lemma 18 in [7]. ◀

4 The Bound Theorem

In this section we prove the Bound Theorem which characterises finiteness of the language of an ADC in terms of the maximum height of an accepted term. The theorem is crucial for the correctness of our finiteness checking algorithm.

▶ **Theorem 7** (Bound theorem)**.** *Let $\mathcal{A}$ be an ADC. $L(\mathcal{A})$ is finite iff all accepted terms have have height strictly smaller than*

$$H(\mathcal{A}) = (e+1) \times |Q| \times 2^{c(\mathcal{A})} \times c(\mathcal{A})! \times (d(\mathcal{A}) + 1)$$

To prove the theorem, we use pumping arguments similar to that in [7]. Instead of pumping downward decreasing the size of an accepted term, however, we need to pump upward increasing the size arbitrarily. The modifications of the arguments are laborious and not trivial, but they follow closely the proofs in [7]. A similar construction may also be found in [9]. In fact, [9, Lemma 5.19] is a generalisation of our Bound Theorem to a broader class of automata, but with a worse, though still exponential, exact bound.

To fully understand this section, some familiarity with [7] is helpful. We try to convey the underlying intuitions, but we don't see it productive to copy proofs or definitions verbatim where no change is necessary.

In contrast to downward pumping in [7] which uses an arbitrary ordering $\gg$ satisfying the requirements of Section 6, for our upward pumping argument we need the strict embedding ordering $\ggg$ on terms.

▶ **Definition 8.** *An* upward pumping *(wrt. the strict embedding ordering $\ggg$) is a replacement $\rho[\rho']_p$ where ρ, ρ' are runs such that the target state of $\rho'(\epsilon)$ is the same as the target of $\rho(p)$ and $\rho[\rho']_p \ggg \rho$.*

The proofs of the generalised pumping lemmas in [7] are divided into two parts: pumping without creating close equalities and pumping without creating equalities (far or close). The argument for pumping without creating close equalities are adapted to upward pumping, but the complex details need to be checked. The argument for pumping without creating equalities is replaced by a simpler argument for upward pumping, because if we can pump upward without creating close equalities then we can increase the size of the pumping arbitrarily to prevent any far equalities from being created.

▶ **Definition 9.** *Given $\mathcal{A} = (Q, Q^f, \Delta)$ and an integer k we set (where e is Euler's number):*

$$g(\mathcal{A}, k) = (e \times k + 1) \times |Q| \times 2^{c(\mathcal{A})} \times c(\mathcal{A})!$$

The following is the main pumping lemma needed in the proof of the Bound Theorem. The proof of this lemma occupies most of this section. It is an analogon of [7, Lemma 19] adapted to upward pumping.

▶ **Lemma 10.** *If ρ is a run of $\mathcal{A}$ and $p_1, \ldots, p_{g(\mathcal{A},k)}$ are positions of ρ such that $\rho|_{p_1} \ggg \ldots \ggg \rho|_{p_{g(\mathcal{A},k)}}$ then there are indices $i_0 < \ldots < i_k$ such that the upward pumping $\rho[\rho|_{p_{i_0}}]_{p_{i_j}}$ does not contain any equality close to p_{i_j}.*

▶ **Definition 11.** *Given $p \in \mathrm{Pos}(\rho)$, the set $\mathrm{cr}(p)$ is defined as the set of all triples (β, π, π') such that there is $p' \in \mathrm{Pos}(\rho)$ with $p'\beta = p$ (i.e. $p' = p/\beta$) and $p \prec p'\pi'$ and $\pi \neq \pi'$ or $\pi' \neq \pi$ is a constraint of $\rho(p')$. See Figure 2.*

The intuition is that $\mathrm{cr}(p)$ indicates all possible places above p at which an equality close to p may be created.

▶ **Fact 12.** *If (p', π, π') is an equality close to p, then there is $(\beta, \pi, \pi') \in \mathrm{cr}(p)$ such that $p'\beta = p$.*

▶ **Fact 13.** *For all $p \in \mathrm{Pos}(\rho)$ we have $\mathrm{cr}(p) \subseteq \mathcal{C}(\mathcal{A})$, and thus $|\mathrm{cr}(p)| \leq c(\mathcal{A})$.*

Similarly to [7] we can extract a subsequence $v_0, \ldots, v_{k_2}$ of $p_1, \ldots, p_{g(\mathcal{A},k)}$ such that $\rho(v_0), \ldots, \rho(v_{k_2})$ all have the same target state and $\mathrm{cr}(v_0) = \ldots = \mathrm{cr}(v_{k_2})$, where $k_2 = (e \times k+1) \times c(\mathcal{A})! - 1$. For this purpose, we first extract a subsequence $u_1, \ldots, u_{k_1}$ of $p_1, \ldots, p_{g(\mathcal{A},k)}$ such that all u_i have the same target state, where $k_1 = \frac{g(\mathcal{A},k)}{|Q|} = (e \times k + 1) \times 2^{c(\mathcal{A})} \times c(\mathcal{A})!$.

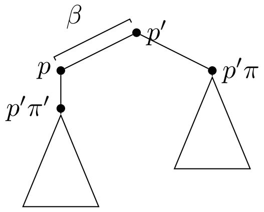

Figure 2 Element of $\mathrm{cr}(p)$.

Because $\mathrm{cr}(p) \subseteq \mathcal{C}(\mathcal{A})$ for each $p \in \mathrm{Pos}(\rho)$, there are at most $2^{c(\mathcal{A})}$ distinct sets $\mathrm{cr}(p)$. Hence, we can extract a subsequence $v_0, \ldots, v_{k_2}$ of $u_1, \ldots, u_{k_1}$ such that $\mathrm{cr}(v_0) = \ldots = \mathrm{cr}(v_{k_2})$ and $k_2 = \frac{k_1}{2^{c(\mathcal{A})}} - 1 = (e \times k + 1) \times c(\mathcal{A})! - 1$.

The idea of the proof of Lemma 10 is illustrated in Figure 3. If for each $j = 1, \ldots, k$ the weak run $\rho[\rho|_{v_0}]_{v_j}$ has a close equality, then (for large enough k_2) there is a (long enough) subsequence $w_1, \ldots, w_m$ of $v_1, \ldots, v_{k_2}$ such that "the same" close equality is created in $\rho[\rho|_{v_0}]_{w_j}$ for each $j = 1, \ldots, m$. We recursively consider the sequence $w_1, \ldots, w_m$ – the number of possible places where a close equality may be created is now smaller – we eliminated one element of $\mathrm{cr}(v_0) = \mathrm{cr}(w_j)$. If $g(\mathcal{A}, k)$ is large enough then we will ultimately eliminate all possible elements of $\mathrm{cr}(v_0)$. Then no close equality can be created in $\rho[\rho|_{v_0}]_{v_j}$ because for each element of $\mathrm{cr}(v_0) = \mathrm{cr}(v_j)$ the subterms at the corresponding positions below v_0 and v_j are identical.

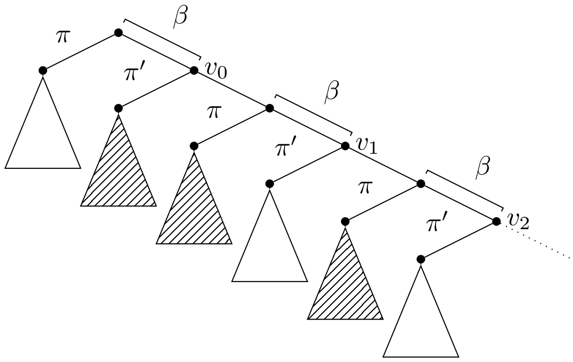

Figure 3 The proof of the pumping lemma.

We proceed with a precise proof.

The *dependency degree* of a subsequence $v_{i_0}, \ldots, v_{i_m}$ is:

$$\mathrm{dep}(v_{i_0} \ldots v_{i_m}) = |\{(\beta, \pi, \pi') \in \mathrm{cr}(v_0) \mid t|_{(v_{i_0}/\beta)\pi} = \ldots = t|_{(v_{i_m}/\beta)\pi}\}|$$

where t is the term associated to ρ.

Let $f(n)$ be the function recursively defined on the interval $[0 \ldots c(\mathcal{A})]$ by:

$$\begin{array}{rcl} f(c(\mathcal{A})) & = & k \\ f(n) & = & (c(\mathcal{A}) - n) \times (f(n+1) + 1) + k - 1 \text{ for } n < c(\mathcal{A}) \end{array}$$

The next lemma is an analogon of Lemma 22 from [7]. It is the main technical lemma needed in the proof of Lemma 10.

▶ **Lemma 14.** *Assume*

($\star$) *for all* $0 \leq j \leq k_2$ *the cardinal of the set* $\{j' \mid k_2 \geq j' > j, \rho[\rho|_{v_j}]_{v_{j'}}$ *has no close equality* $\}$ *is smaller than* k.

Then for all $0 \leq n \leq c(\mathcal{A})$*, there exists a subsequence* $v_{i_0} \ldots v_{i_{f(n)}}$ *of* $v_0 \ldots v_{k_2}$ *such that* $\mathrm{dep}(v_{i_0} \ldots v_{i_{f(n)}}) \geq n$.

Proof. The proof is an adaptation of the proof of Lemma 22 in [7], by induction on n.

The case $n = 0$ is exactly the same as in the proof of Lemma 22 in [7], showing $f(0) \leq k_2$. Let $F(n) = f(c(\mathcal{A}) - n)$ for all $0 \leq n \leq c(\mathcal{A})$. We have:

$$\begin{array}{rcll} F(0) & = & k & \\ F(n) & = & n(F(n-1)+1)+k-1 & \text{for } 1 \leq n \leq c(\mathcal{A}) \end{array}$$

Thus:

$$\begin{array}{rcl} F(n) & = & n! \times (F(0)+1) + k \times \sum_{i=1}^{n} \frac{1}{i!} - 1 \\ & \leq & k \times n! + n! + k \times n! \times (e-1) - 1 \\ & = & n! \times (k \times e + 1) - 1 \end{array}$$

Hence, $f(0) = F(c(\mathcal{A})) \leq c(\mathcal{A})! \times (k \times e + 1) - 1 = k_2$.

For $n+1$, we proceed analogously to [7]. Assume the property is true for $n < c(\mathcal{A})$. By the induction hypothesis, we have a subsequence $v_{i_0} \ldots v_{i_{f(n)}}$ extracted from $v_0 \ldots v_{k_2}$ such that $\mathrm{dep}(v_{i_0} \ldots v_{i_{f(n)}}) \geq n$. By the assumption ($\star$), for at least $f(n) - (k-1) = (c(\mathcal{A}) - n) \times (f(n+1)+1) =: k_3$ positions w among $v_{i_1} \ldots v_{i_{f(n)}}$, the weak run $\rho[\rho|_{v_{i_0}}]_w$ has a close equality (close to w; we take $j = i_0$ in ($\star$) to conclude that there are at least $k_2 - i_0 - (k-1)$ indices j' such that $\rho[\rho|_{v_{i_0}}]_{v_{j'}}$ has a close equality; now $k_2 - i_0 \geq f(n)$ because there exist $f(n)$ indices $i_0 < i_1 < \ldots < i_{f(n)} \leq k_2$). Let $w_1 \ldots w_{k_3}$ be a subsequence of $v_{i_1} \ldots v_{i_{f(n)}}$ consisting of the positions w as above, i.e., for all $j = 1, \ldots, k_3$ the weak run $\rho[\rho|_{v_{i_0}}]_{w_j}$ has a close equality. Hence, for $j = 1, \ldots, k_3$ there exists $(\beta_j, \pi_j, \pi'_j) \in \mathrm{cr}(w_j) = \mathrm{cr}(v_0)$ such that:

- $t|_{(v_{i_0}/\beta_j)\pi'_j} \neq t|_{(v_{i_0}/\beta_j)\pi_j}$,
- $t|_{(v_{i_0}/\beta_j)\pi'_j} = t|_{(w_j/\beta_j)\pi_j}$,

where t is the term associated with ρ. Thus $t|_{(v_{i_0}/\beta_j)\pi_j} \neq t|_{(w_j/\beta_j)\pi_j}$ for $j = 1, \ldots, k_3$.

Because $\mathrm{dep}(v_{i_0} \ldots v_{i_{f(n)}}) \geq n$, there exists a subset $E \subseteq \mathrm{cr}(v_0)$ such that $|E| = n$ and $t|_{(v_{i_0}/\beta)\pi} = \ldots = t|_{(v_{i_{f(n)}}/\beta)\pi}$ for $(\beta, \pi, \pi') \in E$. In particular, $t|_{(v_{i_0}/\beta)\pi} = t|_{(w_1/\beta)\pi} = \ldots = t|_{(w_{k_3}/\beta)\pi}$ for $(\beta, \pi, \pi') \in E$. Hence, $\{(\beta_1, \pi_1, \pi'_1), \ldots, (\beta_{k_3}, \pi_{k_3}, \pi'_{k_3})\} \cap E = \emptyset$ (because $t|_{(v_{i_0}/\beta_j)\pi_j} \neq t|_{(w_j/\beta_j)\pi'_j}$ for $j = 1, \ldots, k_3$). By Fact 13 we have $|\mathrm{cr}(v_0)| \leq c(\mathcal{A})$. Thus, there are at most $c(\mathcal{A}) - n$ distinct tuples among $(\beta_1, \pi_1, \pi'_1), \ldots, (\beta_{k_3}, \pi_{k_3}, \pi'_{k_3})$. Thus there exist $1 \leq j_0 < \ldots < j_{f(n+1)} \leq k_3$ such that $(\beta_{j_0}, \pi_{j_0}, \pi'_{j_0}) = \ldots = (\beta_{j_{f(n+1)}}, \pi_{j_{f(n+1)}}, \pi'_{j_{f(n+1)}})$, because $\frac{k_3}{c(\mathcal{A})-n} = f(n+1)+1$. Let $(\beta', \pi, \pi') = (\beta_{j_0}, \pi_{j_0}, \pi'_{j_0})$ be this tuple. Because $t|_{(w_j/\beta')\pi} = t|_{(v_{i_0}/\beta')\pi}$ for $1 \leq j \leq k_3$, $t|_{(w_{j_0}/\beta')\pi} = \ldots = t|_{(w_{j_{f(n+1)}}/\beta')\pi}$. Since $(\beta', \pi, \pi') \notin E$:

$$\mathrm{dep}(w_{j_0} \ldots w_{j_{f(n+1)}}) > \mathrm{dep}(v_{i_0} \ldots v_{i_{f(n)}}) \geq n.$$

This completes the proof because $w_{j_0} \ldots w_{j_{f(n+1)}}$ is a subsequence of $v_0 \ldots v_{k_2}$. ◀

Proof of Lemma 10. Follows the proof of Lemma 19 in [7]. Assume ($\star$) holds to derive a contradiction. Then for $n = c(\mathcal{A})$ and $f(n) = k$ there exists a subsequence $v_{i_0} \ldots v_{i_k}$ of $v_0 \ldots v_{k_2}$ such that $\mathrm{dep}(v_{i_0} \ldots v_{i_k}) \geq c(\mathcal{A})$. But $|\mathrm{cr}(v_0)| \leq c(\mathcal{A})$ by Fact 13, so for all $(\beta, \pi, \pi') \in \mathrm{cr}(v_0)$ we have $t|_{(v_{i_0}/\beta)\pi} = \ldots = t|_{(v_{i_k}/\beta)\pi}$.

Assume $\rho[\rho|_{v_{i_0}}]_{v_{i_j}}$ has a close equality for some $1 \le j \le k$. There is $(\beta, \pi, \pi') \in \mathrm{cr}(v_{i_j}) = \mathrm{cr}(v_0)$ such that $t|_{(v_{i_0}/\beta)\pi'} \neq t|_{(v_{i_0}/\beta)\pi}$ and $t|_{(v_{i_0}/\beta)\pi'} = t|_{(v_{i_j}/\beta)\pi}$. Hence, $t|_{(v_{i_0}/\beta)\pi} \neq t|_{(v_{i_j}/\beta)\pi}$. Contradiction. Thus, each $\rho[\rho|_{v_{i_0}}]_{v_{i_j}}$ has no close equality for $1 \le j \le k$. Then the cardinality of the set in ($\star$) for $j = 0$ is at least k (note that by definition $k \le k_2$) which contradicts ($\star$).

Thus, ($\star$) cannot hold. This implies that for $1 \le j \le k$ the upward pumping $\rho[\rho|_{v_{i_0}}]_{v_{i_j}}$ does not have a close equality. ◀

▶ **Corollary 15.** *Let ρ be a run of $\mathcal{A}$ and $p_1 \prec \ldots \prec p_{g(\mathcal{A},k)}$ be positions of ρ such that $|p_{j+1}/p_j| > d(\mathcal{A})$ (i.e., the distance between two consecutive positions greater than $d(\mathcal{A})$). Then there exist indices $i_0 < \ldots < i_k$ such that the pumping $\rho[\rho_j^m]_{p_{i_j}}$ for $j > 0$ does not have a close equality for any $m \ge 0$ where: $\rho_j^0 = \rho|_{p_{i_0}}$ and $\rho_j^{m+1} = \rho|_{p_{i_0}}[\rho_j^m]_{p_{i_j}/p_{i_0}}$. See Figure 4.*

Proof. Since $p_1 \prec \ldots \prec p_{g(\mathcal{A},k)}$, we have $\rho|_{p_1} \ggg \ldots \ggg \rho|_{p_{g(\mathcal{A},k)}}$. By Lemma 10 there exist indices $i_0 < \ldots < i_k$ such that $\rho[\rho_j^0]_{p_{i_j}}$ does not have a close equality. Since $|p_{i_j}/p_{i_0}| > d(\mathcal{A})$, noting that $\rho[\rho_j^0]_{p_{i_j}} = \rho[\rho|_{p_{i_0}}[\rho_j^0]_{p_{i_j}/p_{i_0}}]_{p_{i_0}}$, we can prove by induction on m that $\rho[\rho_j^m]_{p_{i_j}}$ has no close equality either. Indeed, any close equality in $\rho[\rho_j^{m+1}]_{p_{i_j}}$ must be a close equality in $\rho|_{p_{i_0}}[\rho_j^m]_{p_{i_j}/p_{i_0}}$, because $|p_{i_j}/p_{i_0}| \ge d(\mathcal{A})$. But then we would have the same close equality in $\rho|_{p_{i_0}}[\rho_j^0]_{p_{i_j}/p_{i_0}}$. ◀

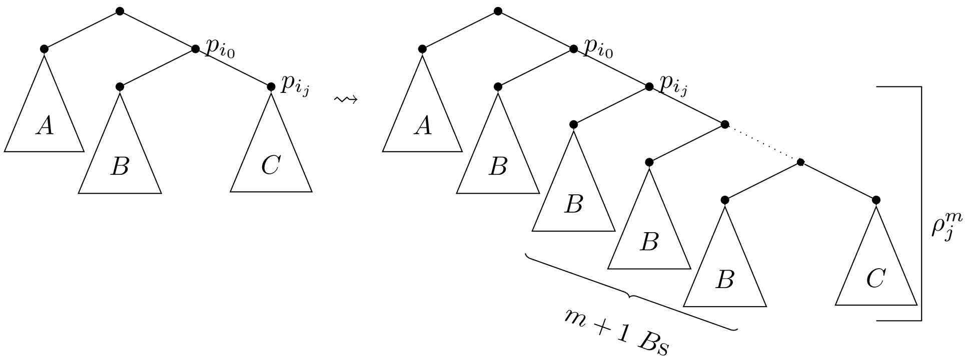

Figure 4 Repeated pumping.

▶ **Corollary 16.** *Under the assumptions of the previous Corollary 15, there exist indices $i_0 < \ldots < i_k$ and $m_0 \ge 0$ such that the pumping $\rho[\rho_j^m]_{p_{i_j}}$ for $j > 0$ and $m \ge m_0$ does not have any equality (close or far).*

Proof. By Corollary 15, $\rho[\rho_j^m]_{p_{i_j}}$ does not have a close equality for any $m \ge 0$. We can choose m to be large enough so that no far equality is created either. Indeed, if an equality (p, π, π') far from p_{i_j} is created, then e.g. $p\pi \preceq p_{i_j}$ and $p\pi' \parallel p_{i_j}$ and $t|_{p\pi} = t|_{p\pi'}$. By making m large enough we can ensure $|\rho_j^m| > |t|_{p'}|$ for any $p' \parallel p_{i_j}$, and then the equality $t|_{p\pi} = t|_{p\pi'}$ is impossible. ◀

Proof of the Bound Theorem 7. If the height of the run is $\ge G(\mathcal{A})$ then we can choose $g(\mathcal{A}, 1)$ positions $p_1, \ldots, p_{g(\mathcal{A},1)}$ satisfying the requirements of Corollary 16. This gives us infinitely many different accepting runs $\rho[\rho_j^m]_{p_{i_j}}$ for $m \ge m_0$. Conversely, if the language is infinite then there can be no bound on the maximal height of an accepting run. ◀

5 Automaton recognising the intersection of a regular tree language with the set of normal forms of a TRS

Given a tree grammar G it is standard to construct a finite tree automaton $\mathcal{A}_G$ recognising the language $L(G)$. See e.g. [5].

The next step is to construct the normal forms ADC $\mathcal{A}_R$ for a given term rewriting system R. The automaton $\mathcal{A}_R$ recognises the ground normal forms of R. The constraints are necessary to handle non-left-linear rules in R. No constraints are generated if R is left-linear.

Finally, we construct the product automaton $\mathcal{A}_G \times \mathcal{A}_R$ which recognises the intersection of $L(\mathcal{A}_G)$ and $L(\mathcal{A}_R)$.

5.1 Construction of the normal forms automaton

The construction of $\mathcal{A}_R$ is described in detail in [7]. We recall it for completeness.

- Let $\mathcal{L}$ be the set of the left-hand sides of R.
- Let $\mathcal{L}_1$ be the subset of the linear terms in $\mathcal{L}$.
- Let $\mathcal{L}_2$ be the set of linearisations of the nonlinear terms in $\mathcal{L}$. For each $l \in \mathcal{L}_2$ we denote its nonlinear origin by $\#l \in \mathcal{L}$.
- Let Q_0 consist of all strict subterms of terms in $\mathcal{L}_1 \cup \mathcal{L}_2$ (modulo renaming of variables) plus two special states:
  - a single variable x which will accept all terms,
  - q_r which will accept only reducible terms of R.

  Note $|Q_0| \leq ||R|| + 2$.
- The set of states Q_R consists of all unifiable subsets of $Q_0 \setminus \{q_r\}$ plus q_r. Each element of Q_R different from q_r is denoted by q_u where u is the term resulting from unifying all elements of the state with the mgu of the state. Note $|Q_R| \leq 2^{|Q_0|} \leq 2^{||R||+2}$.
- Δ_R is the set of all rules of the form

  $$f(q_{u_1}, \ldots, q_{u_n}) \xrightarrow{c} q_u$$

  where $q_{u_1}, \ldots, q_{u_n}, q_u \in Q_R$ and:
  1. if one of the q_{u_i}'s is q_r or $f(u_1, \ldots, u_n)$ is an instance of some $s \in \mathcal{L}_1$, then $q_u = q_r$ and $c = \top$,
  2. otherwise, u is the mgu of all terms $v \in Q_0 \setminus \{q_r\}$ such that $f(u_1, \ldots, u_n)$ is an instance of v, and the constraint c is defined by:

  $$\bigwedge_{\substack{l \in \mathcal{L}_2 \\ u, l \text{ unifiable} \\ f(u_1,\ldots,u_n) \text{ is an instance of } l}} \quad \bigvee_{\substack{x \in \mathrm{Var}(\#l) \\ \#l|_{p_1} = \#l|_{p_2} = x \\ p_1 \neq p_2}} p_1 \neq p_2$$
- Take $\mathcal{A}_R = (Q_R, Q_R \setminus \{q_r\}, \Delta_R)$.

$|Q_R|$ is exponential in R and each constraint has size polynomial in $||R||$.

5.2 Construction of the product automaton

Given two ADCs $\mathcal{A}_1 = (Q_1, Q_1^f, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, Q_2^f, \Delta_2)$, the *product ADC* $\mathcal{A}_1 \times \mathcal{A}_2 = (Q, Q^f, \Delta)$ is defined by:

- $Q = Q_1 \times Q_2$,
- $Q^f = Q_1^f \times Q_2^f$,
- Δ consists of the transitions $f((q_1, q_1'), \ldots, (q_n, q_n')) \xrightarrow{c_1 \wedge c_2} (q, q')$ for every pair of transitions $f(q_1, \ldots, q_n) \xrightarrow{c_1} q \in \Delta_1$ and $f(q_1', \ldots, q_n') \xrightarrow{c_2} q' \in \Delta_2$.

Note that $|Q| = |Q_1| \times |Q_2|$, $|\Delta| = |\Delta_1| \times |\Delta_2|$, $c(\mathcal{A}) \leq c(\mathcal{A}_1) + c(\mathcal{A}_2)$, $s(\mathcal{A}) \leq s(\mathcal{A}_1) + s(\mathcal{A}_2)$, $n(\mathcal{A}) \leq n(\mathcal{A}_1) + n(\mathcal{A}_2)$ and $d(\mathcal{A}) = \max(d(\mathcal{A}_1), d(\mathcal{A}_2))$ (there are no new atomic constraints).

6 Emptiness

To check if $L(G) \cap \mathrm{NF}(R) = \emptyset$, we run the emptiness decision algorithm from [7] on $\mathcal{A}_G \times \mathcal{A}_R$. The algorithm runs in exponential time. For completeness, we give a brief presentation of the emptiness decision algorithm from [7]. The following lemmas and definitions come from [7]. Let $\mathcal{A} = (Q, Q_f, \Delta)$ be the ADC whose emptiness we want to check.

▶ **Definition 17.** *The ordering $\gg$ on terms over Δ is defined by: $\rho_1 \gg \rho_2$ iff $I(\rho_1) > I(\rho_2)$ where $I(\rho)$ is the triple $(\mathrm{depth}(\rho), M(\rho), \rho)$ with $M(\rho)$ the multiset of strict subterms of ρ. The ordering $>$ on triples is the lexicographic product of:*
1. *the ordering on natural numbers,*
2. *the multiset extension of $\gg$ (see e.g. [1, Definition 2.5.3]),*
3. *the lexicographic path order extending a total order on the signature (see e.g. [1, Definition 5.4.12]).*

The lexicographic path order in the third component may be replaced by any reduction order total on ground terms.

▶ **Lemma 18.** *$\gg$ is monotonic, well-founded and total on ground terms. Moreover, if $\mathrm{depth}(\rho) > \mathrm{depth}(\rho')$ then $\rho \gg \rho'$.*

One could replace $\gg$ with any order satisfying the conditions of the above lemma.

▶ **Definition 19** (Emptiness decision algorithm)**.** *Let $E_q^0 = \emptyset$ for each state $q \in Q$. For $m \geq 0$, let E_q^{m+1} consist of all runs $\rho = r(\rho_1, \ldots, \rho_n)$ such that:*
- *$\rho_1, \ldots, \rho_n \in \bigcup_{i=0}^m \bigcup_{q \in Q} E_q^i$,*
- *the target state of ρ is q,*
- *for every $p \in \mathrm{Pos}(\rho) \setminus \mathcal{S}(\mathcal{A})$ with $|p| \leq d(\mathcal{A}) + 1$, there is no sequence of length $b(\mathcal{A})$ of runs $\rho_1', \ldots, \rho_{b(\mathcal{A})}'$ in $\bigcup_{i=0}^m \bigcup_{q \in Q} E_q^i$ such that $\rho|_p \gg \rho_{b(\mathcal{A})}' \gg \ldots \gg \rho_1'$ and $\rho(p)$,$\rho_1'(\epsilon)$,…,$\rho_{b(\mathcal{A})}'(\epsilon)$ all have the same target state and for every $1 \leq j \leq b(\mathcal{A})$ the pumping $\rho[\rho_j']_p$ does not contain any equality close to p.*

After a finite number of iterations, we obtain the saturated set $E^* = \bigcup_{m \geq 0} \bigcup_{q \in Q} E_q^m$. The language of $\mathcal{A}$ is empty iff E^* does not contain an accepting run.

A more detailed pseudocode of the algorithm and the calculation of $b(\mathcal{A})$ may be found in Appendix A. The correctness of the algorithm is proven in [7].

▶ **Theorem 20.** *The emptiness decision algorithm runs in time $O(|\mathcal{A}|^{P_0(s(\mathcal{A}))})$ where P_0 is a polynomial.*

Proof. In [7, Theorem 28] it is shown that the emptiness decision algorithm runs in time $O((|Q| \times |\Delta|)^{P_0'(\mathrm{cs}(\mathcal{A}))})$ where P_0' is a polynomial and $\mathrm{cs}(\mathcal{A})$ is the total size of all constraints in $\mathcal{A}$. A careful analysis of the bounds in Lemma 27 and Sections 5.3.2, 5.3.3 in [7] reveals that the exponent in the running time can actually be made polynomial in $s(\mathcal{A})$ at the expense of introducing the size of the automation $|\mathcal{A}|$ into the base. More precisely, the inequalities in Lemma 27 and Sections 5.3.2 and 5.3.3 in [7] initially the exponent is bounded by a polynomial in $s(\mathcal{A})$ (denoted $c(\mathcal{A})$ in [7]), and only this exponent is then replaced according to the inequality $s(\mathcal{A}) \leq d(\mathcal{A})n(\mathcal{A})$. Instead, one can take a smaller bound on $h(\mathcal{A}, k)$ in Lemma 27, then plug it into the inequalities in Section 5.3.2 to get a bound with an exponent polynomial in $s(\mathcal{A})$ and a different base. The base needs to be $|\mathcal{A}|$ instead of $|Q| \times |\Delta|$, because without $\mathrm{cs}(\mathcal{A})$ in the exponent one cannot remove the $\mathrm{cs}(\mathcal{A})$ factor from the base. ◀

In the above theorem, we need $s(\mathcal{A})$ in the exponent instead of $\mathrm{cs}(\mathcal{A})$ because of the product automaton construction: we have $s(\mathcal{A}_1 \times \mathcal{A}_2) \leq s(\mathcal{A}_1) + s(\mathcal{A}_2)$, but this inequality does not hold for cs. In particular, $\mathrm{cs}(\mathcal{A}_G \times \mathcal{A}_R) = |\Delta_{\mathcal{A}_G}| \times \mathrm{cs}(\mathcal{A}_R)$ while $s(\mathcal{A}_G \times \mathcal{A}_R) = s(\mathcal{A}_R)$.

▶ **Proposition 21** ([5, Theorem 1.7.5]). *The following problem is EXPTIME-hard: given n tree automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$, is $L(\mathcal{A}_1) \cap \ldots \cap L(\mathcal{A}_n)$ empty?*

Note that n is a part of the input, *not* a constant.

▶ **Theorem 22.** *Given n finite tree automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$, there is a polynomial-time construction of a linear term rewriting system R such that:*

$$\mathrm{NF}(R) = \{g(s) \mid s \text{ encodes accepting runs of } \mathcal{A}_1, \ldots, \mathcal{A}_n \text{ on a common term}\}$$

The encoding is such that for each n-tuple of runs there exists exactly one term representing this tuple of runs. In particular:

- $\mathrm{NF}(R) = \emptyset$ *iff* $L(\mathcal{A}_1) \cap \ldots \cap L(\mathcal{A}_n) = \emptyset$,
- $\mathrm{NF}(R)$ *is finite iff* $L(\mathcal{A}_1) \cap \ldots \cap L(\mathcal{A}_n)$ *is finite.*

Proof. The construction of the term rewriting system R is exactly the one from [7, Section 6]. We refer there for details. The statement concerning finiteness (the second point) is not present in [7], but it is easily checked. ◀

▶ **Theorem 23.** *Given a regular tree grammar G and a term rewriting system R, the problem of checking the emptiness of $L(G) \cap \mathrm{NF}(R)$ is EXPTIME-complete. The problem is EXPTIME-hard already for linear R.*

Proof. To decide emptiness of intersection, we construct a finite tree automaton (i.e. an ADC without constraints) $\mathcal{A}_G$ with $L(\mathcal{A}_G) = L(G)$, and the normal forms ADC $\mathcal{A}_R$. Then we check the emptiness of the product $\mathcal{A}_G \times \mathcal{A}_R$. We have $|\mathcal{A}_G| = O(|G|)$ and $|\mathcal{A}_R| = O(2^{||R||})$ and $s(\mathcal{A}_R) = O(P_1(|R|))$ for some polynomial P_1. Then $|\mathcal{A}_G \times \mathcal{A}_R| = O(|G|2^{||R||})$ and $s(\mathcal{A}_G \times \mathcal{A}_R) \leq s(\mathcal{A}_G) + s(\mathcal{A}_R) = s(\mathcal{A}_R) = O(P_1(||R||))$. Constructing $\mathcal{A}_G \times \mathcal{A}_R$ takes time proportional to $|\mathcal{A}_G \times \mathcal{A}_R|$. Hence, by Theorem 20 the entire procedure takes time $O((|G|2^{||R||})^{P_0(P_1(||R||))}) = O(|G|^{P(||R||)})$ for some polynomial P.

EXPTIME-hardness follows from Proposition 21, taking G with $L(G) = \mathcal{T}(\mathcal{F})$ (the set of all ground terms) and the R constructed in Theorem 22. ◀

7 Finiteness

By adapting the arguments of [7] for downward pumping, the Bound Theorem 7 could be refined to provide, in addition to the lower bound, also an exponential upper bound on the height of an accepted term. Then a direct application of the Bound Theorem would yield a 3-EXPTIME algorithm for deciding finiteness: check if there are any terms with height between the two bounds. Instead, we use the Bound Theorem together with the emptiness decision algorithm for ADCs to show that the finiteness problem is in EXPTIME.

▶ **Definition 24.** *For a given $N \in \mathbf{N}$, we define an automaton $\mathcal{A}_N = (Q_N, Q_N^f, \Delta_N)$ recognising the language of all terms of height at least N.*

- $Q_N = \{q_i \mid i \in \{0, \ldots, N\}\}$,
- $Q_N^f = \{q_N\}$,
- Δ_N *consists of the transitions:*
  - $a \to q_0$,
  - $f(q_{i_1}, \ldots, q_{i_n}) \to q_{\min(\max(i_1,\ldots,i_n)+1,N)}$ *for $n > 0$ and all $i_1, \ldots, i_n \in \{0, \ldots, N\}$.*

Intuitively, state q_i indicates that a subterm has height at least i.

▶ **Theorem 25.** *Assume the maximum function symbol arity is a fixed constant. Given a regular tree grammar G and a term rewriting system R, the problem of checking the finiteness of $L(G) \cap \mathrm{NF}(R)$ is EXPTIME-complete. The problem is EXPTIME-hard already for linear R.*

Proof. To decide finiteness in exponential time, we first construct the automaton $\mathcal{A} = \mathcal{A}_G \times \mathcal{A}_R$ like in Theorem 23. Then take $\mathcal{A}' = \mathcal{A} \times \mathcal{A}_N$ with

$$N = H(\mathcal{A}) = (e+1) \times |Q| \times 2^{c(\mathcal{A})} \times c(\mathcal{A})! \times (d(\mathcal{A}) + 1)$$

where $H(\mathcal{A})$ is the function from Theorem 7 and $\mathcal{A}_N$ is the automaton from Definition 24 recognising the language of all terms of height at least N. The language of $\mathcal{A}'$ consists of all terms in $L(G) \cap \mathrm{NF}(R)$ with height at least N. By Theorem 7 the language $L(\mathcal{A}) = L(G) \cap \mathrm{NF}(R)$ is finite iff all terms accepted by $\mathcal{A}$ have height $< N$. Hence, $L(\mathcal{A}') = \emptyset$ iff $L(G) \cap \mathrm{NF}(R)$ is finite. Thus, it suffices to check emptiness of $\mathcal{A}'$ with the algorithm outlined in the previous section.

By the proof of Theorem 23 we have $|\mathcal{A}| = O(|G| 2^{||R||})$. Since $|\mathcal{A}_N| = O(N^\alpha)$ with α a constant depending on the maximum function symbol arity, we obtain $|\mathcal{A}'| = O(|G| 2^{||R||} N^\alpha)$. Also $s(\mathcal{A}') = s(\mathcal{A}) = O(P_1(||R||))$. Hence, by Theorem 20 running the emptiness decision algorithm on $\mathcal{A}$ takes time:

$$\begin{aligned}
&O\left((|G| 2^{||R||} N^\alpha)^{P_0(P_1(||R||))}\right) = \\
&O(|G|^{P_2(||R||)} N^{P_3(||R||)}) = \\
&O(|G|^{P_2(||R||)} (|Q| \times 2^{c(\mathcal{A})} \times c(\mathcal{A})! \times (d(\mathcal{A}) + 1))^{P_3(||R||)}) = \\
&O(|G|^{P_2(||R||)} (|G| 2^{||R||} \times 2^{||R||} \times 2^{||R|| \log(||R||)} \times ||R||)^{P_3(||R||)}) = \\
&O(|G|^{P_2(||R||)} (|G| 2^{P_4(||R||)})^{P_3(||R||)}) = \\
&O(|G|^{P(||R||)})
\end{aligned}$$

where the polynomial P depends on the maximum function symbol arity.

To show EXPTIME-hardness, we reduce from the problem of the finiteness of the intersection of the languages of n tree automata: given n tree automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$, is $L(\mathcal{A}_1) \cap \ldots \cap L(\mathcal{A}_n)$ finite? The reduction follows directly from Theorem 22 (taking G with $L(G) = \mathcal{T}(\mathcal{F})$). It remains to show that the finiteness problem for the intersection of the languages of n tree automata is EXPTIME-hard. We reduce the problem of emptiness of intersection of n tree languages (see Proposition 21).

For an automaton $\mathcal{A} = (Q, Q_f, \Delta)$ over signature Σ we create an automaton $\mathcal{A}' = (Q', Q'_f, \Delta')$ over Σ' such that $L(\mathcal{A})$ is empty iff $L(\mathcal{A}')$ is finite. Each non-nullary symbol $f \in \Sigma$ is in Σ'. For each constant $c \in \Sigma$ we have a unary symbol $c \in \Sigma'$. There is an extra unary symbol $S \in \Sigma' \setminus \Sigma$ and an extra constant $C \in \Sigma' \setminus \Sigma$. We set $Q' = Q \cup \{q_S\}$ and $Q'_f = Q_f$. The transitions Δ' include:

- $f(q_1, \ldots, q_n) \to q$ if it is in Δ and $n > 0$,
- $c(q_S) \to q$ if $c \to q \in \Delta$,
- $C \to q_S$,
- $S(q_S) \to q_S$.

The automaton $\mathcal{A}$ accepts a term t iff $\mathcal{A}'$ accepts terms which result from t by replacing each constant occurrence c with $c(S^k(C))$ for some k (possibly different k for different occurrences). Now, $L(\mathcal{A}_1) \cap \ldots \cap L(\mathcal{A}_n)$ is finite iff $L(\mathcal{A}'_1) \cap \ldots \cap L(\mathcal{A}'_n)$ is finite. Indeed $L(\mathcal{A}'_1) \cap \ldots \cap L(\mathcal{A}'_n)$ contains all terms from $L(\mathcal{A}_1) \cap \ldots \cap L(\mathcal{A}_n)$ with each constant c replaced with $c(S^k(C))$ for some k. ◀

8 Experiments

We evaluate our approach using three examples. The first example is a minimal example inspired by Boolean algebra and highlights the limitations of the approach, as well as some opportunities to overcome them. Examples 2 and 3 extend practical examples from the literature [11]: Example 2 applies the technique to the automatic construction of programs. Example 3 computes paths through a large labyrinth in order to illustrate scalability with linear rewrite systems compared to the SMT-solver based approach in [11]. All examples are available in our Haskell implementation, which accompanies this paper [2].

8.1 Example 1 - Boolean Algebra

Single-sorted Boolean ground terms over a signature containing a binary function symbol **AND** and constants **T**, **F** are recognised by the tree grammar G_B:

$$G_B \quad = \quad (b, \{b\}, \{\mathbf{T}, \mathbf{F}, \mathbf{AND}\}, \{b \to \mathbf{T},\ b \to \mathbf{F},\ b \to \mathbf{AND}(b, b)\})$$

A simple rewrite system can normalize terms by evaluating all function applications of **AND**. One way to specify evaluation rules for **AND** is to use the rewrite system RS_B:

$$RS_B \quad = \quad \{\mathbf{AND}(\mathbf{F}, x) \to \mathbf{F},\ \mathbf{AND}(x, \mathbf{F}) \to \mathbf{F},\ \mathbf{AND}(x, x) \to x\}$$

Using the construction in Section 5.1 yields the normal forms ADC $\mathcal{A}_B = (Q_B, Q_B^f, \Delta_B)$ recognizing $\mathrm{NF}(RS_B)$:

$$\begin{aligned} Q_B \quad &= \quad \{q_0, q_1, q_2\} \qquad Q_B^f = \{q_1, q_2\} \\ \Delta_B \quad &= \quad \{\mathbf{T} \xrightarrow{\top} q_1,\ \mathbf{F} \xrightarrow{\top} q_2,\ \mathbf{AND}(q_1, q_1) \xrightarrow{1 \neq 2} q_1\} \\ &\qquad \cup \{\mathbf{AND}(p_1, p_2) \xrightarrow{\top} q_0 \mid p_1, p_2 \in Q, p_1 \neq q_1 \vee p_2 \neq q_1\} \end{aligned}$$

The language $L(G_B) \cap L(\mathcal{A}_B) = \{\mathbf{T}, \mathbf{F}\}$ is finite and non-empty. In the worst case, the emptiness checking algorithm from Definition 19 needs to enumerate and store at least b terms (if the result is empty), where b is the value computed in Appendix A. For our example, the corresponding values are $b(\mathcal{A}_G \times \mathcal{A}_B) = b_{\text{empty}} = 235018$ for emptiness and $b(\mathcal{A}_G \times \mathcal{A}_B \times \mathcal{A}_N) = b_{\text{fin}} = 7300813834$ for finiteness. Here, the enumeration stops after the first iteration because there exists a term in $L(G_B) \cap L(\mathcal{A}_B)$ of height one. Since $L(G_B) \cap L(\mathcal{A}_B)$ is finite, the finiteness check must enumerate at least b_{fin} terms, which is not practically feasible.

Manual inspection of our example reveals that the rewrite rule $\mathbf{AND}(x, x) \to x$ can be simplified to $\mathbf{AND}(\mathbf{T}, \mathbf{T}) \to \mathbf{T}$, while retaining the same set of normal forms.

$$RS_B^{\text{lin}} \quad = \quad \{\mathbf{AND}(\mathbf{F}, x) \to \mathbf{F},\ \mathbf{AND}(x, \mathbf{F}) \to \mathbf{F},\ \mathbf{AND}(\mathbf{T}, \mathbf{T}) \to \mathbf{T}\}$$

Using this simplification we obtain the ADC $\mathcal{A}_B^{\text{lin}} = (Q_B^{\text{lin}}, Q_B^{f,\text{lin}}, \Delta_B^{\text{lin}})$:

$$\begin{aligned} Q_B^{\text{lin}} &= \{q_0, q_1, q_2\} \qquad Q_B^{f,\text{lin}} = \{q_1, q_2\} \\ \Delta_B &= \{\mathbf{T} \xrightarrow{\top} q_1,\ \mathbf{F} \xrightarrow{\top} q_2\} \cup \{\mathbf{AND}\ p \xrightarrow{\top} q_0 \mid p \in Q \times Q)\} \end{aligned}$$

The automaton $\mathcal{A}_B^{\text{lin}}$ is built for the linear rewrite system RS_B^{lin} and all its disequality constraints are empty (true) by construction, resulting in a finite tree automaton without constraints. Hence, finiteness can be checked in polynomial time wrt. the automaton's size.

Our Haskell implementation exactly matches the expectations from theory: emptiness results are computed immediately (under 1 second on a laptop from 2018 with a 2,7 GHz quad core processor and 16GB Ram). For finiteness we had to abort after over 6 hours in the nonlinear case, while the linear case also computes in under 1 second.

8.2 Example 2 - Construction of sorting functions

Kallat et al. [11] describe how to perform program construction of applications of sorting functions using a tree grammar as an intermediate result of a type inhabitation algorithm. Their grammar (up to renaming of non-terminals) is given as follows:

$$\begin{aligned} G_{\text{sort}} \;=\; & (2, \{0,1,2,3,4\}, \{\mathbf{values}, \mathbf{id}, \mathbf{inv}, \mathbf{sortmap}, \mathbf{min}, \mathbf{default}, \mathbf{@}\}, \\ & \{\ 4 \to \mathbf{@}(\mathbf{@}(\mathbf{sortmap}, 1), 3),\ 2 \to \mathbf{@}(\mathbf{id}, 2),\ 2 \to \mathbf{@}(\mathbf{@}(\mathbf{min}, 0), 4), \\ & 0 \to \mathbf{@}(\mathbf{id}, 0),\ 0 \to \mathbf{default},\ 0 \to \mathbf{@}(\mathbf{inv}, 0),\ 0 \to \mathbf{@}(\mathbf{@}(\mathbf{min}, 0), 4), \\ & 1 \to \mathbf{id},\ 1 \to \mathbf{inv},\ 3 \to \mathbf{@}(\mathbf{id}, 3),\ 3 \to \mathbf{values}\}) \end{aligned}$$

Evaluation rules can be stated as the following rewrite system:

$$\begin{aligned} RS_{\text{sort}} \;=\; & \{\mathbf{@}(\mathbf{id}, x) \to x,\ \mathbf{@}(\mathbf{inv}, \mathbf{@}(\mathbf{inv}, x)) \to x, \\ & \mathbf{@}(\mathbf{@}(\mathbf{sortmap}, x), \mathbf{@}(\mathbf{@}(\mathbf{sortmap}, y), z)) \to \mathbf{@}(\mathbf{@}(\mathbf{sortmap}, x), z) \\ & \mathbf{@}(\mathbf{@}(\mathbf{min}, \mathbf{@}(\mathbf{@}(\mathbf{min}, x), y)), y) \to \mathbf{@}(\mathbf{@}(\mathbf{min}, x), y)\} \end{aligned}$$

The normal form ADC $\mathcal{A}_{\text{sort}}$ has 26 reachable states and 47 transitions (after reduction of non-reachable states). We obtain bound values $b(\mathcal{A}_G \times \mathcal{A}_{RS}) = b_{\text{empty}} = 4655986860$ and $b(\mathcal{A}_G \times \mathcal{A}_{RS} \times \mathcal{A}_N) = b_{\text{fin}} = 445281079421917\,88$. The language $L(G_{\text{sort}}) \cap L(\mathcal{A}_{\text{sort}})$ is finite and non-empty. Since the smallest term has a height of four, the emptiness checking algorithm terminates after four iterations with result *False* (non-empty). The algorithm for deciding finiteness needs to enumerate and store at least b_{fin} terms before terminating with result *True* (finite).

In [11] no rewrite rules are used. Instead, the authors construct constraints that forbid terms of form $@(\mathbf{id}, x)$, $@(\mathbf{inv}, x)$, and $@(\mathbf{min}, @(x, y))$, declaring these forms as non-normal without providing replacements (i.e. the right-hand sides of the rewrite rules). Our approach is flexible enough to do the same, since right-hand sides of the rewrite system are ignored. We may use the following linear rules:

$$RS_{\text{sort}}^{\text{lin}} \;=\; \{@(\mathbf{id}, x) \to x,\ @(\mathbf{inv}, x) \to x,\ @(\mathbf{min}, @(x, y)) \to x\}$$

The normal forms automaton $\mathcal{A}_{\text{sort}}^{\text{lin}}$ is a finite tree automaton (no disequality constraints). Our Haskell implementation can check emptiness immediately and finiteness again is only possible in the linear case, with results being available in under one second.

8.3 Example 3 - Filtering redundant paths in a labyrinth

The last example in [11] is a grammar for finding paths through a labyrinth. In this grammar non-terminals are **up**, **down**, **left**, **right**, **start**. Rules ensure that only valid paths can be constructed. The example is scaled for randomly generated labyrinths. Redundant paths, such as $\mathbf{up}(\mathbf{down}(x))$ get filtered. The authors of [11] note that their SMT-solver based approach only scales up to labyrinths with 10×10 fields. Reproducing these experiments, we ran the CLS framework to generate labyrinth solution grammars for up to 30×30 fields (stopping there to limit the runtime of CLS). Using the four rewrite rules

$$RS_{\text{lab}} = \{\mathbf{up}(\mathbf{down}(x)) \to x,\ \mathbf{down}(\mathbf{up}(x)) \to x,\ \mathbf{left}(\mathbf{right}(x)) \to x,\ \mathbf{right}(\mathbf{left}(x)) \to x\}$$

our Haskell implementation again produces immediate emptiness results for the intersection, while finiteness is computed in under 5 minutes. The reduced intersection automaton has 7619 reachable states, 8956 transitions and a value $b(\mathcal{A}_G \times \mathcal{A}_{RS} \times \mathcal{A}_N) = b_{\text{fin}} = 149360346492$. This result is a true improvement over scalability issues encountered in solver-based solutions.

9 Conclusion

We have shown that the emptiness and finiteness problems of the intersection $L(G) \cap \mathrm{NF}(R)$ of the language of a regular tree grammar G and the normal forms of a rewrite system R are EXPTIME-complete. Both problems are practically relevant for enumerating terms generated by the CLS synthesis algorithm (and potentially other synthesis approaches). Enumeration can be implemented by bottom-up enumerating all terms of $L(G)$ and filtering them according to membership in $\mathrm{NF}(R)$. Without the decision procedures the enumeration algorithm does not know when to (not) stop: in the empty case it does not need to enumerate anything. In the finite case, it needs to enumerate until all terms of height N (as computed in the proof of Theorem 25) are listed. In the infinite case, it can continue to enumerate.

We have also conducted practical experiments, which show that, although also EXPTIME-complete, the problems are feasible for left-linear rewrite systems. Results for the nonlinear case, however, were less encouraging, since here the proposed algorithm always has to enumerate a very large set of terms before being able to decide emptiness and an even larger set before being able to decide finiteness. It is an interesting goal for future research to investigate other algorithms, which might perform better in the average case. Also, heuristics that are incomplete (e.g., with bounds on the probability of obtaining a decision) are an interesting area for future research.

References

1. Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1999.
2. J. Bessai, L. Czajka, F. Laarmann, and J. Rehof. Restricting tree grammars with rewriting. `https://github.com/FelixLaarmann/tree-grammar`, 2022.
3. Jan Bessai. *A type-theoretic framework for software component synthesis*. PhD thesis, Technical University of Dortmund, Germany, 2019. URL: `http://hdl.handle.net/2003/38387`.
4. Jan Bessai, Tzu-Chun Chen, Andrej Dudenhefner, Boris Düdder, Ugo de'Liguoro, and Jakob Rehof. Mixin composition synthesis based on intersection types. *Logical Methods in Computer Science*, 14(1), 2018. `doi:10.23638/LMCS-14(1:18)2018`.
5. Hubert Comon, Max Dauchet, Remi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications*. INRIA, 2008.

6 Hubert Comon and Florent Jacquemard. Ground reducibility is EXPTIME-complete. In *LICS 1997*, pages 26–34, 1997.

7 Hubert Comon and Florent Jacquemard. Ground reducibility is EXPTIME-complete. *Inf. Comput.*, 187(1):123–153, 2003.

8 Boris Düdder, Moritz Martens, Jakob Rehof, and Pawel Urzyczyn. Bounded combinatory logic. In Patrick Cégielski and Arnaud Durand, editors, *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*, volume 16 of *LIPIcs*, pages 243–258. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPIcs.CSL.2012.243.

9 Guillem Godoy and Omer Giménez. The HOM problem is decidable. *J. ACM*, 60(4):23:1–23:44, 2013.

10 Fadil Kallat, Carina Mieth, Jakob Rehof, and Anne Meyer. Using component-based software synthesis and constraint solving to generate sets of manufacturing simulation models. In *CIRP*, pages 556–561. Elsevier, 2020. doi:10.1016/j.procir.2020.03.018.

11 Fadil Kallat, Tristan Schäfer, and Anna Vasileva. CLS-SMT: bringing together combinatory logic synthesis and satisfiability modulo theories. In Giselle Reis and Haniel Barbosa, editors, *Proceedings Sixth Workshop on Proof eXchange for Theorem Proving, PxTP 2019, Natal, Brazil, August 26, 2019*, volume 301 of *EPTCS*, pages 51–65, 2019. doi:10.4204/EPTCS.301.7.

12 Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas W. Reps. Semantics-guided synthesis. *Proc. ACM Program. Lang.*, 5(POPL):1–32, 2021. doi:10.1145/3434311.

13 Patrick Landwehr and Christof Löding. Tree Automata with Global Constraints for Infinite Trees. In Rolf Niedermeier and Christophe Paul, editors, *36th International Symposium on Theoretical Aspects of Computer Science (STACS 2019)*, volume 126 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 47:1–47:15, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

14 Parthasarathy Madhusudan. Synthesizing reactive programs. In Marc Bezem, editor, *Computer Science Logic, 25th International Workshop / 20th Annual Conference of the EACSL, CSL 2011, September 12-15, 2011, Bergen, Norway, Proceedings*, volume 12 of *LIPIcs*, pages 428–442. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011. doi:10.4230/LIPIcs.CSL.2011.428.

15 Hitoshi Ohsaki. Beyond regularity: Equational tree automata for associative and commutative theories. In Laurent Fribourg, editor, *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*, volume 2142 of *Lecture Notes in Computer Science*, pages 539–553. Springer, 2001.

16 Hitoshi Ohsaki and Hiroyuki Seki. Languages modulo normalization. In Boris Konev and Frank Wolter, editors, *Frontiers of Combining Systems, 6th International Symposium, FroCoS 2007, Liverpool, UK, September 10-12, 2007, Proceedings*, volume 4720 of *Lecture Notes in Computer Science*, pages 221–236. Springer, 2007.

17 Jakob Rehof and Pawel Urzyczyn. Finite combinatory logic with intersection types. In *TLCA*, volume 6690 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 2011. doi:10.1007/978-3-642-21691-6_15.

18 Jakob Rehof and Moshe Y. Vardi. Design and synthesis from components (dagstuhl seminar 14232). *Dagstuhl Reports*, 4(6):29–47, 2014. doi:10.4230/DagRep.4.6.29.

19 Tristan Schäfer, Jim A. Bergmann, Rafael G. Carballo, Jakob Rehof, and Petra Wiederkehr. A synthesis-based tool path planning approach for machining operations. In *CIRP*, pages 918–923. Elsevier, 2021. doi:10.1016/j.procir.2021.11.154.

20 Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

21 Sigrid Wenzel, Jana Stolipin, Jakob Rehof, and Jan Winkels. Trends in automatic composition of structures for simulation models in production and logistics. In *WSC*, pages 2190–2200. IEEE, 2019. doi:10.1109/WSC40007.2019.9004959.

A The emptiness algorithm

In the emptiness decision algorithm, we use the following value

$$b(\mathcal{A}) = \max(\beta k + \gamma, |Q| \times |\mathcal{F}|)$$

where

- $s = s(\mathcal{A})$ is the number of *distinct* suffixes of positions π, π' in an atomic constraint $\pi \neq \pi'$ in a rule of $\mathcal{A}$.
- $n = n(\mathcal{A})$ is the *maximum* number of atomic constraints in a rule of $\mathcal{A}$,
- $d = d(\mathcal{A})$ is the maximum length of π or π' in an atomic constraint $\pi \neq \pi'$ in a rule of $\mathcal{A}$,
- $e = \sum_{i=1}^{s} \frac{1}{i!}$,
- $\beta = (d+1)n(e|Q|2^s s! + 1)$,
- $\gamma = (2dne + 1)(d+1)n|Q|2^s s!$,
- $k = \lceil \frac{\beta+\sqrt{\beta^2+4\gamma}}{2} \rceil$.

The value $b(\mathcal{A})$ above is a slight improvement on [7] where a less precise bound is used. For Lemma 26 in [7], the value k must satisfy $k^2 \geq h(\mathcal{A}, k)$ where

$$\begin{array}{rcl} h(\mathcal{A}, k) & = & (d+1)n(k + g(\mathcal{A}, k + 2dn)) \\ g(\mathcal{A}, k) & = & (ek+1)|Q|2^s s! \end{array}$$

One can calculate that

$$\begin{array}{rcl} h(\mathcal{A}, k) & = & (d+1)nk + (d+1)n((e(k+2dn)+1)|Q|2^s s!) \\ & = & (d+1)nk + (d+1)n(ek|Q|2^s s! + 2dne|Q|2^s s! + |Q|2^s s!) \\ & = & k(d+1)n(e|Q|2^s s! + 1) + (2dne+1)(d+1)n|Q|2^s s! \\ & = & \beta k + \gamma \end{array}$$

Hence, we need to find the smallest k such that

$$k^2 - \beta k - \gamma \geq 0$$

The least integer equal or greater than the second (positive) root $\frac{\beta+\sqrt{\beta^2+4\gamma}}{2}$ of the quadratic equation does the job, and we obtain the k listed above. According to the proofs in [7], we can then take $b(\mathcal{A}) = \max(h(\mathcal{A}, k), |Q| \times |\mathcal{F}|)$.

The pseudocode for the emptiness decision algorithm is presented in Listing 1.

Listing 1 Emptiness decision algorithm.

```
Input: 𝒜 = (Q, Q^f, Δ).
Output: true iff L(𝒜) = ∅.

Let C be the set of suffixes of positions π, π' in atomic
constraints of transition rules in Δ.
E* ← ∅
M ← ∅
repeat
  E ← ∅
  for all r ∈ Δ do
    Let m be the arity of r (i.e. the arity of the top
    symbol in the rule).
    for all ρ_1, ..., ρ_m ∈ E* s.t. r(ρ_1, ..., ρ_m) is a run do
      ρ ← r(ρ_1, ..., ρ_m)
      if ρ ∈ M then
        continue
      endif
      M ← M ∪ {ρ}
      v ← true
      for all p ∈ Pos(ρ) \ C s.t. |p| ≤ d + 1 do
        for all ρ'_1, ..., ρ'_b ∈ E* s.t. all ρ'_i(ε) have the
                same target state as ρ(p)
        do
          if ρ|_p ≫ ρ'_b ≫ ... ≫ ρ'_1 and
             for all 1 ≤ j ≤ b, ρ[ρ'_j]_p does not contain any
             equality close to p
          then
            v ← false
          endif
        done
      done
      if v then
        E ← E ∪ {ρ}
      endif
    done
  done
  E* ← E* ∪ E
until E = ∅
if E* contains an accepting run then
  return false
else
  return true
endif
```