

On the Separability Problem of String Constraints

Parosh Aziz Abdulla

Uppsala University, Sweden
parosh@it.uu.se

Mohamed Faouzi Atig

Uppsala University, Sweden
mohamed_faouzi.atig@it.uu.se

Vrunda Dave

IIT Bombay, India
vrunda@cse.iitb.ac.in

Shankara Narayanan Krishna

IIT Bombay, India
krishnas@cse.iitb.ac.in

Abstract

We address the separability problem for straight-line string constraints. The separability problem for languages of a class C by a class S asks: given two languages A and B in C , does there exist a language I in S separating A and B (i.e., I is a superset of A and disjoint from B)? The separability of string constraints is the same as the fundamental problem of interpolation for string constraints. We first show that regular separability of straight line string constraints is undecidable. Our second result is the decidability of the separability problem for straight-line string constraints by piece-wise testable languages, though the precise complexity is open. In our third result, we consider the positive fragment of piece-wise testable languages as a separator, and obtain an EXPSPACE algorithm for the separability of a useful class of straight-line string constraints, and a PSPACE-hardness result.

2012 ACM Subject Classification Security and privacy → Logic and verification; Theory of computation → Verification by model checking

Keywords and phrases string constraints, separability, interpolants

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

The *string* data type is widely used in almost all modern programming and scripting languages. Many of the well-known security vulnerabilities such as SQL injections and cross-site scripting attacks are often caused by an improper handling of strings. The detection of such vulnerabilities is usually reduced to the satisfiability of a formula which is then solved by SMT solvers (e.g., [41, 42, 48, 32]). Therefore, string constraints solving has received considerable attention in recent years (e.g. [13, 12, 29, 48, 49, 44, 30, 28, 2, 32, 10, 26]) and this has led to the development of many efficient string solvers such as HAMPI [29], Z3-str3 [9], CVC4 [30, 31, 39], S3P [44, 45], Trau [2, 3, 5], SLOTH [26] and OSTRICH [14].

In spite of these advances, most of these tools do not provide any completeness guarantees. The foundational question regarding the decidability of string solving for a large class of string constraints has several challenges to be overcome. A major difficulty is that any reasonably expressive class of string constraints is either undecidable, or has its decidability status open for several years [21, 22, 23]. In fact, the satisfiability problem is undecidable even for the class of string constraints with concatenation (useful to model assignments in the program) and transduction (useful to model sanitisation and replacement operations) [14]. A direction of research is to find meaningful and expressive subclasses of string constraints for which the satisfiability problem is decidable (e.g., [4, 5, 22, 32, 26, 12]). An interesting



© Parosh Aziz Abdulla, M. Faouzi Atig, V. Dave, S. Krishna;
licensed under Creative Commons License CC-BY

Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

subclass, that has been studied extensively, is that of straight-line (SL) string constraints (e.g., [26, 14, 32, 26, 12]). The SL fragment was introduced by Barceló and Lin in [32]. Roughly, an SL constraint models the feasibility of a path of a string-manipulating program that can be generated by symbolic execution. The *satisfiability* of the SL fragment was shown to be EXPSPACE-complete in [32] and forms the basis of many of the tools above [26, 12].

In this paper, we focus on the fundamental problem of *interpolation/separability* for the SL fragment of string constraints. An *interpolant* for a pair of formulas A, B is a formula over their common vocabulary that is implied by A and is inconsistent with B . The Craig-Lyndon interpolation technique is very well-known in mathematical logic. McMillan [33] in his pioneering work, has also recognized interpolation as an efficient method for automated construction of abstractions of systems. Interpolation based algorithms have been developed for a number of problems in program verification [33, 34, 35].

Interpolation procedures have been implemented by many solvers for the theories most commonly used in program verification like linear arithmetic, uninterpreted functions with equality and some combination of such theories. In most of these algorithms, the interpolants were simple. The interpolation technique can also be used to check the unsatisfiability. In fact, the existence of an interpolant for formulas A and B implies the unsatisfiability of $A \wedge B$.

The notion of *separators* in formal language theory is the counterpart of interpolants in logic. The separability problem for languages of a class \mathcal{C} by a class \mathcal{S} asks: given two languages $I, E \in \mathcal{C}$, does there exist a language $S \in \mathcal{S}$ separating I and E ? That is, $I \subseteq S$ and $S \cap E = \emptyset$. The language S is called the separator of I, E . Separability is a classical problem of fundamental interest in theoretical computer science, and has recently received a lot of attention. For instance, regular separability has been studied for one-counter automata [17], Parikh automata [15], and well-structured transition systems [18]. In the following, we use the terms interpolant or separator of two SL string constraints to mean the same thing, since the solutions of a string constraint can be interpreted as a language.

In this paper, we first show that any string constraint ϕ can be written as the conjunction of two SL string constraints A and B . Therefore, the interpolation problem for the pair A and B can be used to check the unsatisfiability of the string constraint ϕ . (Recall that the satisfiability problem for general string constraints is undecidable [14].)

Then, we consider the regular separability problem for SL string constraints. We show that this problem is undecidable (Theorem 2) by a reduction from the halting problem of Turing Machines. The main technical difficulty here is to ensure that the encoding of a sequence of configurations of a Turing machine results in SL string constraints.

Due to this undecidability, we focus on the separability problem of SL string constraints by piece-wise testable languages (PTL). A PTL is a finite Boolean combination of special regular languages called *piece languages* of the form $\Sigma^* a_1 \Sigma^* a_2 \dots \Sigma^* a_n \Sigma^*$, where all $a_j \in \Sigma$. PTL is a very natural and well-studied class of languages in the context of the separability problem (e.g. [37, 19, 20]). Furthermore, among the various separator classes considered in the literature, the class of piecewise testable languages (PTL) seems to be the most tractable: PTL-separability of regular languages is in PTIME [37, 19]. To decide the PTL-separability of SL string constraints, we first encode the solutions of an SL string constraint as the language of an Ordered Multi-Pushdown Automaton (OMPA) (Section 4.1). Then, we show that the PTL-separability of SL constraints can be reduced to the PTL-separability of OMPAs. To show the decidability of the latter problem, we first prove that the language of an OMPA: (1) is a full trio [24] and (2) has a semilinear Parikh image. Using (1), we obtain the equivalence of the PTL separability problem and the diagonal problem for OMPAs from [20], where the

can also be derived from the fact
that OMPA are a special case of HOPDA
of level 2

equivalence has been shown to hold for full trios. Next, the decidability of PTL-separability problem for OMPAs is obtained from the decidability of the diagonal problem for OMPAs: the latter is obtained using (2) and [20] where the decidability of the diagonal problem has been shown for languages having a semilinear Parikh image. As a corollary of these results, we obtain the decidability of the PTL-separability problem for SL string constraints and OMPAs; however the exact complexity is still an open question. In fact, it is an open problem in the case of OMPAs with one stack (i.e., Context-Free Languages (CFLs)) [20].

Given the complexity question, we propose the class of positive piecewise testable languages (PosPTL) as separators. PosPTL is obtained as a negation-free Boolean combination of piece languages. As a first result (Theorem 12) we show that deciding PosPTL-separability for any language class has a very elegant proof: it suffices to check if the upward (downward) closure of one of the languages is disjoint from the other language. Using this result, we prove the PSPACE-completeness of the PosPTL-separability for CFLs, thereby progressing on the complexity front with respect to a problem which is open in the case of PTL-separability for CFLs. Then, we focus on a class of SL string constraints where the variables used in outputs of the transducers are independent of each other. This class contains SL string constraints with functional transducers (computing partial functions, by associating at most one output with each input). We prove the decidability and EXPSPACE membership for the PosPTL-separability of this class by first encoding the solutions of string constraints as outputs of two way transducers (2NFT), and then proving the decidability of PosPTL-separability for 2NFT.

Due to lack of space, missing proofs of the technical results can be found in the appendix.

Related work. The satisfiability problem for string constraints is an active research area and there is a lot of progress in the last decade (e.g., [38, 29, 32, 12, 14, 5, 22, 23, 4, 47]). An interpolation based semi-decision procedure for string constraints has been proposed in [4]. As far as we know, this is the first time the separability problem has been studied in the context of string constraints.

2 Preliminaries

Notations. Let $[i, j]$ denote the set $\{i, \dots, j\}$ for $i, j \in \mathbb{N}$. Let Σ be a finite alphabet. Σ^* denotes the set of all finite words over Σ and $\Sigma^+ = \Sigma^* \setminus \{\epsilon\}$ where ϵ is the empty word. We denote $\Sigma \cup \{\epsilon\}$ by Σ_ϵ . Let $u \in \Sigma^*$. We use u^R to denote the reverse of u . The length of the word u is denoted $|u|$ and the i^{th} symbol of u by $u[i]$. Given two words $u \in \Sigma^*$ and $v \in \Sigma^*$, we say that u is a subword of v (denoted $u \preceq v$) if there is a mapping $h : [1, |u|] \mapsto [1, |v|]$ such that (1) $u[i] = v[h(i)]$ for all $i \in [1, |u|]$, and (2) $h(i) < h(j)$ for all $i < j$.

(Multi-tape)-Automata. A *Finite State Automaton* (FSA) over an alphabet Σ is a tuple $\mathcal{A} = (Q, \Sigma, \delta, I, F)$, where Q is a finite set of *states*, $\delta \subseteq Q \times \Sigma_\epsilon \times Q$ is a set of *transitions*, and $I \subseteq Q$ (resp. $F \subseteq Q$) are the *initial* (resp. *accepting*) states. \mathcal{A} accepts a word w iff there is a sequence $q_0 a_1 q_1 a_2 \dots a_n q_n$ such that $(q_{i-1}, a_i, q_i) \in \delta$ for all $1 \leq i \leq n$, $q_0 \in I$, $q_n \in F$, and $w = a_1 \dots a_n$. The *language* of \mathcal{A} , denoted $\mathcal{L}(\mathcal{A})$, is the set all accepted words.

Given $n \in \mathbb{N}$, a *n-tape automaton* \mathcal{T} is an automaton over the alphabet $(\Sigma_\epsilon)^n$. It recognizes the relation $\mathcal{R}(\mathcal{T}) \subseteq (\Sigma^*)^n$ that contains the n -tuple of words (w_1, w_2, \dots, w_n) for which there is a word $(a_{(1,1)}, a_{(2,1)}, \dots, a_{(n,1)}) \dots (a_{(1,m)}, a_{(2,m)}, \dots, a_{(n,m)}) \in \mathcal{L}(\mathcal{T})$ with $w_i = a_{(i,1)} \dots a_{(i,m)}$ for all $i \in \{1, \dots, n\}$. A *transducer* is a 2-tape automaton.

Well-quasi orders. Given a (possibly infinite set) C , a quasi-order on C is a reflexive and transitive relation $\sqsubseteq \subseteq C \times C$. An infinite sequence c_1, c_2, \dots in C is said to be saturating if there exists indices $i < j$ s.t. $c_i \sqsubseteq c_j$. A quasi-order \sqsubseteq is said to be a well-quasi order

(wqo) on C if every infinite sequence in C is saturating. Observe that the subword ordering \preceq between words u, v over a finite alphabet Σ is well-known to be a wqo on Σ^* [25].

Upward and Downward Closure. Given a wqo \sqsubseteq on a set C , a set $U \subseteq C$ is said to be upward closed if for every $a \in U$ and $b \in C$, with $a \sqsubseteq b$, we have $b \in U$. The upward closure of a set $U \subseteq C$ is defined as $U\uparrow = \{b \in C \mid \exists a \in U, a \sqsubseteq b\}$. It is known that every upward closed set U can be characterized by a finite *minor*. A minor $M \subseteq U$ is s.t. (i) for each $a \in U$, there is a $b \in M$ s.t. $b \sqsubseteq a$, and (ii) for all $a, b \in M$ s.t. $a \preceq b$, we have $a = b$. For an upward closed set U , let \min be the function that returns the minor of U . Downward closures are defined analogously. The downward closure of a set $D \subseteq C$ is defined as $D\downarrow = \{b \in C \mid \exists a \in D, b \sqsubseteq a\}$. The notion of subword relation and thus upward and downward closures naturally extends to n -tuples of words. The subword relation here is component wise i.e. $(u_1, \dots, u_n) \preceq_n (v_1, \dots, v_n)$ iff $u_i \preceq v_i$ for all $i \in [1, n]$.

String Constraints. An atomic string constraint φ over an alphabet Σ and a set of string variables \mathcal{X} is either: (1) a *membership constraint* of the form $x \in \mathcal{L}(\mathcal{A})$ where $x \in \mathcal{X}$ and \mathcal{A} is a FSA (i.e., the evaluation of x is in the language of a FSA \mathcal{A} over Σ), or (2) a *relational constraint* of the form $(t', t) \in \mathcal{R}(\mathcal{T})$ where t and t' are string terms (i.e., concatenation of variables in \mathcal{X}) and \mathcal{T} is a transducer over Σ , and t and t' are related by a relation recognised by the transducer \mathcal{T} . $(t', t) \in \mathcal{R}(\mathcal{T})$ can also be written as $t' = \mathcal{T}(t)$, that is, \mathcal{T} produces t' as the output on input t . For a given term t , $|t|$ denotes the number of variables appearing in t .

A string constraint Ψ is a conjunction of atomic string constraints. We define the semantics of string constraints using a mapping η , called *evaluation*, that assigns for each variable a word over Σ . The evaluation η can be extended in the straightforward manner to string terms as follows $\eta(t_1 \cdot t_2) = \eta(t_1) \cdot \eta(t_2)$. We extend also η to atomic constraints as follows: (1) $\eta(x \in \mathcal{L}(\mathcal{A})) = \top$ iff $\eta(x) \in \mathcal{L}(\mathcal{A})$, and (2) $\eta((t, t') \in \mathcal{R}(\mathcal{T})) = \top$ iff $(\eta(t), \eta(t')) \in \mathcal{R}(\mathcal{T})$.

The truth value of Ψ for an evaluation η is defined in the standard manner. If $\eta(\Psi) = \top$ then η is a *solution* of Ψ , written $\eta \models \Psi$. The formula Ψ is *satisfiable* iff it has a solution.

A string constraint is said to be *Straight Line*¹ (SL) if it can be rewritten as $\Psi' \wedge \bigwedge_{i=1}^k \varphi_i$ where Ψ' is a conjunction of membership constraints, and $\varphi_1, \dots, \varphi_k$ are relational constraints such that (1) there is a sequence of different string variables x_1, x_2, \dots, x_n with $n \geq k$, and (2) φ_i is of the form $(x_i, t_i) \in \mathcal{R}(\mathcal{T}_i)$ such that if a variable x_j is appearing in t_i then $j > i$. A string constraint in the SL form is called an SL formula. Observe that any string formula can be rewritten as a conjunction of two SL formulas (by using extra-variables).

► **Lemma 1.** *Given a string constraint Ψ , it is possible to construct two SL string constraints Ψ_1 and Ψ_2 such that Ψ is satisfiable iff $\Psi_1 \wedge \Psi_2$ is satisfiable.*

Let Ψ be a string constraint and x_1, \dots, x_n be the set of variables appearing in Ψ . We use $\mathcal{L}(\Psi)$ to denote the language of Ψ which consists of the set of n -tuple of words (u_1, \dots, u_n) such that there is an evaluation η with $\eta(\Psi) = \top$ and $\eta(x_i) = u_i$ for all $i \in [1, n]$.

The Separability Problem. Given two classes of languages \mathcal{C} and \mathcal{S} , the separability problem for \mathcal{C} by the separator class \mathcal{S} is defined as follows: Given two languages I and E from the class \mathcal{C} , does there exist a separator $S \in \mathcal{S}$ such that $I \subseteq S$ and $E \cap S = \emptyset$.

¹ In [32], the authors consider Boolean combinations of membership constraints. Our results can be extended to handle this. In [32], they consider also constraints of the form $x = t$. Such constraints can be encoded using our relational constraints.

3 Regular Separability of String Constraints

Let Σ be an alphabet and k, n be two natural numbers. A set R of n -tuples of words over Σ is said to be regular (REG) iff there is a sequence of finite-state automata $\mathcal{A}_{(i,1)}, \dots, \mathcal{A}_{(i,n)}$ for every $i \in [1, k]$ such that $R = \bigcup_{i=1}^k [\mathcal{L}(\mathcal{A}_{(i,1)}) \times \dots \times \mathcal{L}(\mathcal{A}_{(i,n)})]$. The REG separability problem for string constraints consists in checking for two given string constraints Ψ and Ψ' over the string variables x_1, \dots, x_n whether there is a regular set $R \subseteq (\Sigma^*)^n$ such that $\mathcal{L}(\Psi) \subseteq R$ and $R \cap \mathcal{L}(\Psi') = \emptyset$. The regular separability problem is undecidable in general. This can be seen as an immediate corollary of the fact that the satisfiability problem of string constraints is undecidable [36, 14] even for a simple formula of the form $(x, x) \in \mathcal{R}(\mathcal{T})$ where \mathcal{T} is a transducer and x is a string variable. To see why, consider Ψ to be $(x, x) \in \mathcal{R}(\mathcal{T})$ and Ψ' such that $\mathcal{L}(\Psi') = \Sigma^*$. It is easy to see that Ψ' and Ψ are separable by a regular set iff Ψ is unsatisfiable. In the following, we show a stronger result, namely that this undecidability still holds even for REG separability between two SL formulas.

► **Theorem 2.** *The REG separability problem is undecidable even for SL string constraints.*

4 PTL-Separability of String Constraints

Given the undecidability of REG separability, we focus on the separability problem using piece-wise testable languages (PTL). We show that the problem is in general undecidable and then we show its decidability in the case of SL formulas. The undecidability proof is exactly the same as in the case of the REG separability (since Σ^* is a PTL) while the decidability proof is done by reduction to its corresponding problem for the class of Ordered Multi Pushdown Automata (OMPA) [7, 11] (which we show its decidability). In the rest of this section, we first recall the definition of PTL and extend it to n -tuples of words. Then, we define the class of OMPAs and show the decidability of its separability problem by PTL. Finally, we show the decidability of the separability problem for SL formulas by PTL.

Piece-wise testable languages. Let Σ be an alphabet. A piece-language is a regular language of the form $\Sigma^* a_1 \Sigma^* a_2 \Sigma^* \dots \Sigma^* a_k \Sigma^*$ where $a_1, a_2, \dots, a_k \in \Sigma$. The class of piecewise testable languages (PTL) is defined as a finite Boolean combination of piece languages [43]. We can define PTL for an n -tuple alphabet with $n \in \mathbb{N}$, as follows: The class of PTL over n -tuple words (denoted n -PTL) is defined as the finite Boolean combination of languages of the form $(\Sigma^*)^n \mathbf{v}_1 (\Sigma^*)^n \dots (\Sigma^*)^n \mathbf{v}_k (\Sigma^*)^n$ where $\mathbf{v}_i \in (\Sigma_\epsilon)^n$ for all $i \in [1, k]$.

Ordered Multi Pushdown Automata. Let Σ be a finite alphabet and $n \geq 1$ a natural number. Ordered multi-pushdown automata extend the model of pushdown automata with multiple stacks. An n -Ordered Multi Pushdown Automaton (OMPA or n -OMPA) is a tuple $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, Q_0, F)$ where (1) Q, Q_0 and F are finite sets of states, initial states and final states, respectively, (2) Γ is the stack alphabet and it contains the special symbol \perp , and (3) δ is the transition relation. OMPA are restricted in a sense that pop operations are only allowed from the first non-empty stack. A transition in δ is of the form $(q, \perp, \dots, \perp, A_j, \epsilon, \dots, \epsilon) \rightarrow^a (q', \gamma_1, \dots, \gamma_n)$ where $A_j \in \Gamma_\epsilon$ represents the symbol that will be popped from the stack j on reading the input symbol $a \in \Sigma_\epsilon$, and $\gamma_i \in \Gamma^*$ represents the sequence of symbols which is going to be pushed on the stack i . The condition that $A_1 = \dots = A_{j-1} = \perp$ (resp. $A_{j+1} = \dots = A_n = \epsilon$) corresponds to the fact that the stacks $1, \dots, j-1$ (resp. $j+1, \dots, n$) are required to be empty (resp. inaccessible).

A configuration of \mathcal{A} is of the form $(q, w, \alpha_1, \dots, \alpha_n)$ where $q \in Q$, $w \in \Sigma^*$ and $\alpha_1, \dots, \alpha_n \in (\Gamma \setminus \{\perp\})^* \cdot \{\perp\}$. The transition relation \rightarrow between the set of configurations of \mathcal{A} is defined as follows: Given two configurations $(q, w, \alpha_1, \dots, \alpha_n)$ and

$(q', w', \alpha'_1, \dots, \alpha'_n)$, we have $(q, w, \alpha_1, \dots, \alpha_n) \rightarrow (q', w', \alpha'_1, \dots, \alpha'_n)$ iff there is a transition $(q, A_1, \dots, A_n) \rightarrow^a (q', \gamma_1, \dots, \gamma_n) \in \delta$ such that $w = aw'$ and $\alpha'_i = \gamma_i u_i$ where $\alpha_i = A_i u_i$ for all $i \in [1, n]$. We use \rightarrow^* to denote the transitive and reflexive closure of \rightarrow . A word $w \in \Sigma^*$ is accepted by \mathcal{A} if there exists a sequence of configurations c_1, \dots, c_m such that: (1) c_1 is of the form $(q_0, w, \perp, \dots, \perp)$, with $q_0 \in Q_0$, (2) c_m is of the form $(q_f, \epsilon, \perp, \dots, \perp)$, with $q_f \in F$, and (3) $c_i \rightarrow c_{i+1}$ for all $i \in [1, m-1]$. The language of \mathcal{A} (denoted by $\mathcal{L}(\mathcal{A})$) is defined as the set of words accepted by \mathcal{A} . The languages accepted by OMPA are referred to as OMPL.

In the following, we show that the separability problem for OMPL by PTL is decidable. As a first step, we show that the class of OMPL forms a *full trio* [24, 20]. We first recall the definition of a full-trio. Let L be a language over an alphabet A , and let $B \subseteq A$. The B -projection of a word $w \in A^*$ is the longest scattered subword containing only symbols from B . For example, if $A = \{a, b, c\}$, $B = \{b, c\}$, then the B -projection of $w = ababac$ is bbc . The B -upward closure of L is the set of all words that can be obtained by taking a word in L and padding it with symbols from B . For example, if $L = \{w\}$ for w as above, then the B -upward closure of L is the set $B^*aB^*bB^*aB^*bB^*aB^*cB^*$. A class of languages \mathcal{C} is a full trio if it is effectively closed under (1) B -projection for every finite alphabet B , (2) B -upward closure for every finite alphabet B , and (3) intersection with regular languages.

► **Lemma 3.** *The class of OMPLs forms a full trio.*

To connect the PTL separability problem of SL string constraints to that of OMPL, we first use lemma 4. Lemma 4 states that the PTL separability problem for OMPL is equivalent to the *diagonal problem* for OMPL. We recall the diagonal problem [20]. Fix a class of languages \mathcal{C} as above and a language $L \in \mathcal{C}$ over alphabet $\Sigma = \{a_1, \dots, a_n\}$. Assume an ordering $a_1 < \dots < a_n$ on Σ . For $a \in \Sigma$ and $w \in L$, let $\#_a(w)$ denote the number of occurrences of a in w . The *Parikh image* of w is the n -tuple $(\#_{a_1}(w), \dots, \#_{a_n}(w))$. The *Parikh image* of L is the set of all Parikh images of words in L . An n -tuple $(m_1, \dots, m_n) \in \mathbb{N}^n$ is dominated by another n -tuple (d_1, \dots, d_n) iff $m_i \leq d_i$ for all $1 \leq i \leq n$. The *diagonal problem* for \mathcal{C} is the decision problem, which, given as input, a language L from \mathcal{C} asks whether each n -tuple $(m, \dots, m) \in \mathbb{N}^n$ is dominated by some Parikh image of L .

► **Lemma 4.** *The PTL-separability and diagonal problems are equivalent for OMPLs.*

Proof. This equivalence has been shown for full trios in [20] (see Lemma 3). ◀

► **Lemma 5.** *Each language L in OMPL has a semilinear Parikh image.*

► **Theorem 6.** *Given two OMPAs \mathcal{A}_1 and \mathcal{A}_2 , checking whether there is a PTL L such that $\mathcal{L}(\mathcal{A}_1) \subseteq L$ and $L \cap \mathcal{L}(\mathcal{A}_2) = \emptyset$ is decidable.*

Proof. The proof follows from Lemmas 5, 4 and [20], from where we know that the diagonal problem is decidable for classes of languages having semilinear Parikh images. ◀

► **Remark 7.** For the case of 1-OMPA, the PTL separability problem is already known to be decidable [20] but its complexity is still an open problem.

4.1 From SL formula to OMPA

In the following, we show that the n -PTL separability problem for SL formulas can be reduced to the PTL separability problem for OMPLs. To that aim, we proceed as follows: First, we show how to encode an n -tuple of words $(\in (\Sigma^*)^n)$ as a word over $(\Sigma \cup \{\#\})^*$. Then, we show how to encode the set of solutions of an atomic relational constraint $(x, t) \in \mathcal{R}(\mathcal{T})$ using

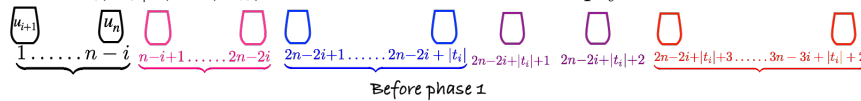
the stacks of an OMPA. Finally, we construct an OMPA that accepts exactly the language of a given SL formula Ψ . This construction will make use of the constructed OMPAs that encode the set of atomic relational constraints appearing in Ψ . Let Σ be an alphabet.

Encoding an n -tuple of words. Let n be a natural number. We assume w.l.o.g. that the special symbol $\#$ does not belong to Σ . We define the function **Encode** that maps any n -tuple word $\mathbf{w} = (w_1, \dots, w_n) \in (\Sigma^*)^n$ to the word $w_1\#w_2\#\dots\#w_n$.

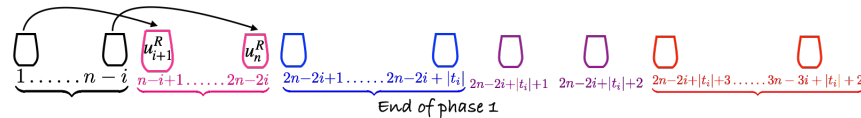
From SL atomic relational constraints to OMPAs. Let x_1, x_2, \dots, x_n be a sequence of string variables. Let P_i be a relational constraint of the form $(x_i, t_i) \in \mathcal{R}(\mathcal{T}_i)$ such that if a variable x_j is appearing in the term t_i , then $j > i$. In the following, we show that we can construct an OMPA \mathcal{A}_i with $(3n + |t_i| + 2 - 3i)$ stacks such that if \mathcal{A}_i starts with a configuration where the first $(n - i)$ stacks contain, respectively, the evaluations $\eta(x_{i+1}), \dots, \eta(x_n)$ (and all the other stacks are empty), then it can compute an evaluation $\eta(x_i)$ of the variable x_i such that: (1) $(\eta(x_i), \eta(t_i)) \in \mathcal{R}(\mathcal{T}_i)$ and the evaluations $\eta(x_i), \dots, \eta(x_n)$ are stored in the last $n - i + 1$ stacks of \mathcal{A}_i . Such an OMPA \mathcal{A}_i will be used as a gadget when constructing the OMPA \mathcal{A} that accepts exactly the language of a given SL formula Ψ .

► **Lemma 8.** *We can construct an OMPA $\mathcal{A}_i = (Q_i, \Sigma, \{\perp\} \cup \Sigma, \delta_i, \{q_i^{init}\}, \{q_i^{final}\})$ with $(3n + |t_i| + 2 - 3i)$ -stacks such that for every $u_i, \dots, u_n \in \Sigma^*$, we have $(q_i^{init}, \epsilon, u_{i+1}\perp, \dots, u_n\perp, \perp, \dots, \perp) \rightarrow^* (q_i^{final}, \epsilon, \perp, \dots, \perp, u_i\perp, u_{i+1}\perp, \dots, u_n\perp)$ iff $(\eta(x_i), \eta(t_i)) \in \mathcal{R}(\mathcal{T}_i)$ with $\eta(x_j) = u_j$ for all $j \in [i, n]$.*

Proof. In the proof, we omit the input ϵ from the OMPA configurations, and only write the state, and stack contents. Let us assume that the string term t_i is of the form $y_1y_2\dots y_{|t_i|}$. Observe that $y_j \in \{x_{i+1}, \dots, x_n\}$. The OMPA \mathcal{A}_i proceeds in phases starting from the configuration $(q_i^{init}, u_{i+1}\perp, \dots, u_n\perp, \perp, \dots, \perp)$. To begin, stacks 1 to $n - i$ contain u_{i+1}, \dots, u_n , the evaluations of x_{i+1}, \dots, x_n , and all other stacks are empty. The computation proceeds in 4 phases. The stacks indexed $1, \dots, n - i$ and $n - i + 1, \dots, 2n - 2i$ will be used in the first phase below. The second phase uses stacks indexed $n - i + 1, \dots, 2n - 2i$ and $2n - 2i + 1, \dots, 2n - 2i + |t_i|$ along with the last $n - i$ stacks indexed $2n - 2i + |t_i| + 3$ to $3n - 3i + |t_i| + 2$. In the third phase, stacks indexed $2n - 2i + 1, \dots, 2n - 2i + |t_i|, 2n - 2i + |t_i| + 1$ are used. In the last phase, stacks indexed $2n - 2i + |t_i| + 1$ and $2n - 2i + |t_i| + 2$ are used. At the end of the 4 phases, stacks indexed $2n - 2i + |t_i| + 2, \dots, 3n - 3i + |t_i| + 2$ hold the evaluations of x_i, x_{i+1}, \dots, x_n , and all other stacks are empty.



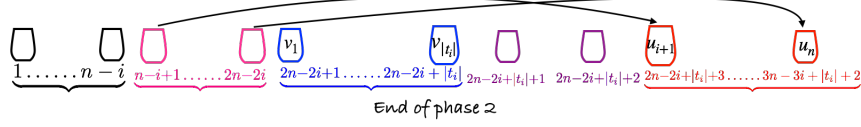
Phase 1. The OMPA \mathcal{A}_i pops the symbols, one by one, from the first $(n - i)$ -stacks $1, \dots, n - i$ and pushes them into the stacks from index $(n - i + 1)$ to $(2n - 2i)$, respectively. At the end of this phase, the new configuration of the OMPA \mathcal{A}_i is $(q_i^{init}, \perp, \dots, \perp, u_{i+1}^R\perp, \dots, u_n^R\perp, \perp, \dots, \perp)$. That is, stacks $n - i + 1, \dots, 2n - 2i$ have u_{i+1}^R, \dots, u_n^R , while all other stacks are empty.



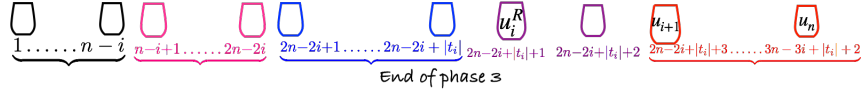
Phase 2. We do two things. (1) the contents of the $n - i$ stacks $n - i + 1, \dots, 2n - 2i$ are moved (in reverse) into the $n - i$ stacks $2n - 2i + |t_i| + 3, \dots, 3n - 3i + |t_i| + 2$. This results in the stacks $2n - 2i + |t_i| + 3, \dots, 3n - 3i + |t_i| + 2$ containing u_{i+1}, \dots, u_n . (2) If y_j appearing in t_i is the variable $x_{i+\ell}$, then the content of stack $n - i + \ell$ (with $n - i + 1 \leq n - i + \ell \leq 2n - 2i$) is also moved (in reverse) to stack $2n - 2i + j$, $1 \leq j \leq |t_i|$. This results in stack $2n - 2i + j$

XX:8 On the Separability Problem of String Constraints

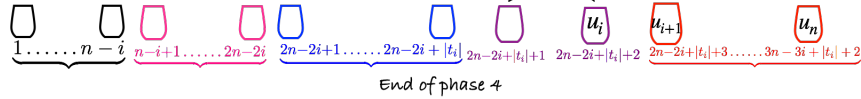
containing $u_{i+\ell}$. Thus, at the end of (1), (2), the stacks $n-i+1, \dots, 2n-2i$ are empty, the stack $2n-2i+|t_i|+\ell+2$ contains $u_{i+\ell}$, the evaluation of $x_{i+\ell}$ for $\ell \geq 1$, while stack $2n-2i+k$ for $1 \leq k \leq |t_i|$ contains u_{i+m} if $y_k = x_{i+m}$. The two stacks $2n-2i+|t_i|+1$ and $2n-2i+|t_i|+2$ are empty at the end of this phase. Stack contents of $2n-2i+k$, $1 \leq k \leq |t_i|$ are referred to as v_k in the figure.



Phase 3. The OMPA \mathcal{A}_i mimics the transducer \mathcal{T}_i . The current state of \mathcal{A}_i is the same as the current state of the simulated transducer. Each transition of \mathcal{T}_i of the form $(q, (a, b), q')$ is simulated by (1) moving the state of \mathcal{A}_i from q to q' , (2) pushing the symbol a into the stack $(2n-2i+|t_i|+1)$, and (3) popping the symbol b from the first non-empty stack having an index between $2n-2i+1$ to $2n-2i+|t_i|$. Recall that the stacks $2n-2i+1$ to $2n-2i+|t_i|$ contain the evaluations of $y_1, \dots, y_{|t_i|}$, for $(\eta(x_i), \eta(y_1) \cdot \eta(y_2) \cdot \dots \cdot \eta(y_{|t_i|})) \in \mathcal{R}(\mathcal{T}_i)$. When the current state of \mathcal{A}_i is in a final state of \mathcal{T}_i and the stacks from index $2n-2i+1$ to $2n-2i+|t_i|$ are empty, then we know that $\eta(y_1) \dots \eta(y_{|t_i|})$ is indeed related by \mathcal{T}_i on $\eta(x_i)$. Then \mathcal{A}_i changes its state to q_i^{final} .



The Last Phase. At the end of the third phase, the current configuration of \mathcal{A}_i is $(q_i^{final}, \perp, \dots, \perp, u_i^R, \perp, u_{i+1}\perp, \dots, u_n\perp)$ such that $(u_i, v_1 \dots v_{|t_i|}) \in \mathcal{R}(\mathcal{T}_i)$: that is, the last $n-i$ stacks $2n-2i+|t_i|+3, \dots, 3n-3i+|t_i|+2$ contain u_{i+1}, \dots, u_n , and stack $2n-2i+|t_i|+1$ contains the reverse of u_i . Then, \mathcal{A}_i pops, one-by-one, the symbols from the $(2n-2i+|t_i|+1)$ -th stack and pushes them, in the reverse order, into the stack $(2n-2i+|t_i|+2)$. Thus, the new configuration of \mathcal{A}_i is of the form $(q_i^{final}, \perp, \dots, \perp, u_i\perp, u_{i+1}\perp, \dots, u_n\perp)$ such that $(u_i, v_1 \dots v_{|t_i|}) \in \mathcal{R}(\mathcal{T}_i)$ where $v_j = u_\ell$ if $y_j = x_\ell$ for $1 \leq j \leq |t_i|$.



From SL formula to OMPAs. In the following, we first construct an OMPA that accepts the encoding of the set of solutions of an SL formula.

► **Lemma 9.** *Given an SL formula Ψ , with x_1, \dots, x_n as its set of variables, it is possible to construct an OMPA \mathcal{A} such that $\mathcal{L}(\mathcal{A}) = \text{Encode}(\mathcal{L}(\Psi))$.*

Proof. Let us assume that Ψ is of the form $\bigwedge_{i=1}^n x_i \in \mathcal{L}(\mathcal{A}_i) \wedge \bigwedge_{i=1}^k \varphi_i$ where $\varphi_1, \dots, \varphi_k$ are relational constraints such that φ_i is of the form $(x_i, t_i) \in \mathcal{R}(\mathcal{T}_i)$. The OMPA \mathcal{A} will have $(n-k + \sum_{i=1}^k (2n-2i+2+|t_i|))$ stacks. \mathcal{A} first guesses an evaluation for the variables x_{k+1}, \dots, x_n in the first $n-k$ stacks and then starts simulating the OMPA \mathcal{A}_k (see Lemma 8 for the definition of \mathcal{A}_k) in order to compute a possible evaluation of the variable x_k such that the relational constraint $(x_k, t_k) \in \mathcal{R}(\mathcal{T}_k)$ holds for that evaluation. After this step, the stacks from index $(2n-2k+|t_k|+2)$ to $(3n-3k+|t_k|+2)$ contain the evaluation of the string variables x_k, \dots, x_n , and all remaining stacks are empty. Now \mathcal{A} can start the simulation of the OMPA \mathcal{A}_{k-1} (Lemma 8) in order to compute a possible evaluation of the variable x_{k-1} such that $(x_k, t_k) \in \mathcal{R}(\mathcal{T}_k) \wedge (x_{k-1}, t_{k-1}) \in \mathcal{R}(\mathcal{T}_{k-1})$ holds for that evaluation. At the start of the simulation of \mathcal{A}_{k-1} by \mathcal{A} , the $n-k+1$ stacks (indexed $(2n-2k+|t_k|+2)$ to $(3n-3k+|t_k|+2)$) contain the evaluations of x_k, \dots, x_n , and the next $2n-2(k-1)+|t_{k-1}|+2$

stacks are used to simulate phases 2-4 of \mathcal{A}_{k-1} . At the end of this, the $n - k + 2$ stacks backwards from the stack indexed $(3n - 3k + |t_k| + 2) + 2n - 2(k - 1) + |t_{k-1}| + 2$ contain the evaluations of x_{k-1}, \dots, x_n . Now, \mathcal{A} simulates $\mathcal{A}_{k-2}, \dots, \mathcal{A}_n$ in the same way. At the end of this simulation phase, the last n -stacks of \mathcal{A} contain an evaluation of the string variables x_1, \dots, x_n that satisfies $\bigwedge_{i=1}^k \varphi_i$. Let us assume that the current configuration of \mathcal{A} at the end of this is of the form $(q^{final}, \perp, \dots, \perp, u_1\perp, u_2\perp, \dots, u_n\perp)$. Then, \mathcal{A} starts popping, one-by-one, from the n -th stack from the last and outputs the read stack symbol $\in \Sigma$ while ensuring that the evaluation u_1 of x_1 belongs to $\mathcal{L}(\mathcal{A}_1)$. When the n -th stack from the last is empty, \mathcal{A} outputs the special symbol $\#$. Then, \mathcal{A} does the same for the i -th stack from last, with $i \in [1, n - 1]$, which contains the evaluation of x_{i+1} . If \mathcal{A} succeeds to empty all stacks, then this means that the evaluation η which associates to the variable x_i , the word u_i for all $i \in [1, n]$ satisfies $\bigwedge_{i=1}^n x_i \in \mathcal{L}(\mathcal{A}_i)$. Hence, $u_1\#u_2\#\dots\#u_n$ is accepted by \mathcal{A} iff $\eta \models \Psi$. ◀

The following lemma shows that the PTL-separability problem for SL formulas can be reduced to the PTL-separability problem for OMPLs.

► **Lemma 10.** *Let Ψ_1 and Ψ_2 be two SL formulae with x_1, \dots, x_n as their set of variables. Let \mathcal{A}_1 and \mathcal{A}_2 be two OMPAs such that $\mathcal{L}(\mathcal{A}_1) = \text{Encode}(\mathcal{L}(\Psi_1))$ and $\mathcal{L}(\mathcal{A}_2) = \text{Encode}(\mathcal{L}(\Psi_2))$. Ψ_1, Ψ_2 are n -PTL separable iff $\mathcal{A}_1, \mathcal{A}_2$ are PTL-separable.*

As an immediate corollary of Theorem 6, Lemma 10, we obtain our main result:

► **Theorem 11.** *The n -PTL separability problem of SL formulae is decidable.*

5 PosPTL-Separability of String Constraints

In this section, we address the separability problem for string constraints by a sub-class of PTL, called positive piece-wise testable languages (PosPTL). A language is in PosPTL iff it is defined as a finite positive Boolean combination (i.e., union and intersection but no complementation) of piece-languages. Given a natural number $n \in \mathbb{N}$, this definition can naturally be extended to n -tuples of words in the straightforward manner (as in the case of PTL) to obtain the class of n -PosPTL. In the following, we first provide a necessary and sufficient condition for the n -PosPTL separability problem of any two languages.

► **Theorem 12.** *Two languages I and E are n -PosPTL separable iff $I \uparrow \cap E = \emptyset$ iff $I \cap E \downarrow = \emptyset$.*

The rest of this section is structured as follows: First, we show that the PosPTL separability is decidable for OMPLs; in the particular case of CFLs, this problem is PSPACE-complete. Then, we use the encoding of SL formulas to OMPAs (as defined in Section 4), and show that n -PosPTL separability of SL formulas reduces to the PosPTL-separability of corresponding OMPLs. Finally, we consider the PosPTL-separability problem for a subclass of SL formulas, called *right sided* SL formulas. We show that the PosPTL separability problem for this subclass is PSPACE-HARD and is in EXPSPACE.

5.1 PosPTL separability of SL formulas

First, we show that the PosPTL separability for OMPLs is decidable.

► **Theorem 13.** *PosPTL separability of OMPLs is decidable.*

Proof. Consider \mathcal{C} to be the class of OMPLs in Theorem 12. Let I and E be two languages belonging to \mathcal{C} as stated in Theorem 12. Then, the set $\min(I\uparrow)$ is effectively computable as an immediate consequence of the *Generalized Valk-Jantzen* construction [6]. The main idea behind this construction is to start with an empty minor set M (so to begin, $M \subseteq I$) and keep adding new words $w \in I$ to M if w is not already in $M\uparrow$. Before adding a new word, we need to test that $I \cap \overline{M\uparrow} \neq \emptyset$ (the complement of $M\uparrow$ intersects with I). This test is decidable since (i) OMPLs are closed under intersection with regular languages and (ii) the emptiness problem for OMPA is decidable. At each step, we remove all the non-minimal words from M (since M is finite). The algorithm terminates due to the Higman's Lemma [25] (the minor of an upward closed set is finite). When the algorithm terminates, $I \subseteq M\uparrow$ and thus $I\uparrow \subseteq M\uparrow$. By construction, $M \subseteq I$ and $M\uparrow \subseteq I\uparrow$. Thus, $M\uparrow = I\uparrow$. Since M is a minor set, we have $\min(I\uparrow) = M$. Using (i) and (ii), we obtain the decidability of checking the emptiness of $I\uparrow \cap E$, and thus PosPTL separability of OMPL is decidable. ◀

As mentioned in section 4, the complexity of PTL-separability for 1-OMPL is open; however, we show that the PosPTL separability problem for 1-OMPL is PSPACE-COMPLETE.

► **Theorem 14.** *The PosPTL-separability for CFLs is PSPACE-COMPLETE.*

For the decidability of the n -PosPTL separability of SL formulas, we use the encoding of SL formulas to OMPAs (as defined in section 4), and show that the n -PosPTL separability of SL formulas reduces to the PosPTL separability of their corresponding OMPLs. The decidability of the n -PosPTL separability of SL formulas follows from Theorem 13.

► **Lemma 15.** *Given two SL formulas Ψ and Ψ' , with x_1, \dots, x_n as their set of variables. Let \mathcal{A} and \mathcal{A}' be two OMPAs such that $\mathcal{L}(\mathcal{A}) = \text{Encode}(\mathcal{L}(\Psi))$ and $\mathcal{L}(\mathcal{A}') = \text{Encode}(\mathcal{L}(\Psi'))$. Then, Ψ and Ψ' are separable by an n -PosPTL iff \mathcal{A} and \mathcal{A}' are separable by a PosPTL.*

As an immediate corollary of Lemma 13 and 15, we obtain the following theorem:

► **Theorem 16.** *The n -PosPTL separability problem of SL formulae is decidable.*

5.2 PosPTL separability of right-sided SL formula

Unfortunately, the proof of Theorem 16 does not allow us to extract any complexity result. Therefore, we consider in this subsection a useful fragment of SL formulas, called *right-sided* SL formulas. Roughly speaking, an SL formula Ψ is *right-sided* iff any variable appearing on the right-side of a relational constraint can not appear on the left-side of any relational constraint. Let us formalize the notion of right-sided SL formulas. Let us assume an SL formula Ψ of the form $\bigwedge_{i=1}^n x_i \in \mathcal{L}(\mathcal{A}_i) \wedge \bigwedge_{i=1}^k (x_i, t_i) \in \mathcal{R}(\mathcal{T}_i)$ with x_1, \dots, x_n as set of variables. Then, Ψ is said to be *right-sided* if none of the variables x_1, \dots, x_k appear in any of t_1, \dots, t_k . We call x_{k+1}, \dots, x_n (resp. x_1, \dots, x_k) *independent* (resp. *dependent*) variables. Observe that the class of SL formulas with functional transducers can be rewritten as right-sided SL formulas. A transducer \mathcal{T} is functional if for every word w , there is at most one word w' such that $(w', w) \in \mathcal{R}(\mathcal{T})$ (\mathcal{T} computes a function). An example of a functional transducer is the one implementing the identity relational constraint (allowing to express the equality $x = t$).

In the following, we show that the PosPTL-separability problem for right-sided SL formulas is in EXPSpace. To show this result, we will reduce the PosPTL-separability problem for right-sided SL formulas to its corresponding problem for two-way transducers.

Two way transducers. Let Σ be a finite input alphabet and let \vdash, \dashv be two special symbols not in Σ . We assume that every input string $w \in \Sigma^*$ is presented as $\vdash w \dashv$, where \vdash, \dashv serve

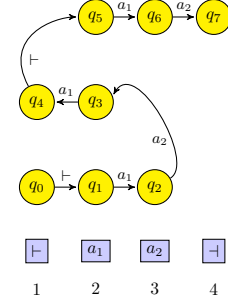
as left and right delimiters that appear nowhere else in w . We write $\Sigma_{\vdash\vdash} = \Sigma \cup \{\vdash, \dashv\}$. A two-way automaton $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ has a finite set of states Q , subsets $I, F \subseteq Q$ of initial and final states and a transition relation $\delta \subseteq Q \times \Sigma_{\vdash\vdash} \times Q \times \{-1, 1\}$. The -1 represents that the reading head moves to left after taking the transition while a 1 represents that it moves to right.

The reading head cannot move left when it is on \vdash , and cannot move right when it is on \dashv . A configuration of \mathcal{A} on reading $w' = \vdash w \dashv$ is represented by (q, i) where $q \in Q$ and i is a position in the input, $1 \leq i \leq |w| + 2$, which will be read in state q . An initial configuration is of the form $(q_0, 1)$ with $q_0 \in I$ and the reading head on \vdash . If $w' = w_1 a w_2$ and the current configuration is $(q, |w_1| + 1)$, and $(q, a, q', -1) \in \delta$, then there is a transition from the configuration $(q, |w_1| + 1)$ to $(q', |w_1|)$ (hence $a \neq \vdash$). Likewise, if $(q, a, q', 1) \in \delta$, we obtain a transition from $(q, |w_1| + 1)$ to $(q', |w_1| + 2)$. A run of \mathcal{A} on reading $\vdash w \dashv$ is a sequence of transitions; it is accepting if it starts in an initial configuration and ends in a configuration of the form $(q, |w| + 2)$ with $q \in F$ and the reading head on \dashv . The language of \mathcal{A} (denoted $\mathcal{L}(\mathcal{A})$) is the set of all words $w \in \Sigma^*$ s.t. \mathcal{A} has an accepting run on $\vdash w \dashv$.

We extend the definition of a two-way automaton $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ into a two-way transducer (2NFT) $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, I, F)$ where Γ is a finite output alphabet. The transition relation is defined as a *finite* subset $\delta \subseteq Q \times \Sigma_{\vdash\vdash} \times Q \times \Gamma^* \times \{-1, 1\}$. The output produced on each transition is appended to the right of the output produced so far. \mathcal{A} defines a relation $\mathcal{R}(\mathcal{A}) = \{(w, u) \mid w \text{ is the output produced on an accepting run of } u\}$. The acceptance condition is the same as in two-way automata. Sometimes, we use the macro-notation $(p, a, q, \alpha, 0)$ to denote a sequence of consecutive transitions (p, a, s, α, d) and (s, b, q, ϵ, d') in δ with $d + d' = 0$, $b \in \Sigma_{\vdash\vdash}$ and s is an extra intermediary state of \mathcal{A} that is not used anywhere else (and that we omit from the set of states of \mathcal{A}).

PosPTL separability of 2NFT. In the following, we study the PosPTL separability of 2NFT. We define here the notion of *visiting sequences* (similar to *crossing sequences* of 2NFT [8]), which will be used in the proof of Lemma 17. Let $w = \vdash a_1 \dots a_n \dashv$ be an input word and let ρ be a run of the 2NFT on w . A visiting sequence at a position x in a run ρ of w captures the states visited in order in the run, each time the reading head is on position x , along with the information pertaining to the direction of the outgoing transition from that state.

For example, in run ρ , if position x is visited for the first time in state q , and the outgoing transition chosen in ρ from q during that visit had direction $+1$, then q^+ will be the first entry in the visiting sequence. For a run ρ , the visiting sequence at a position x is defined as the tuple $\rho|x = (q_1^{d_1}, q_2^{d_2}, \dots, q_h^{d_h})$ of states that have, in order, visited position x in ρ , and whose outgoing transitions had direction d_1, \dots, d_h . In the example, the visiting sequence at position 2 is (q_1^+, q_3^-, q_5^+) , while those at 1 and 3 respectively are (q_0^+, q_4^+) and (q_2^-, q_6^+) .



► **Lemma 17.** *Given a 2NFT \mathcal{T} , if (v, u) is in $\min(\mathcal{R}(\mathcal{T})\uparrow)$ then $|u|$ and $|v|$ are of at most exponential length in the size of \mathcal{T} .*

Proof. In the following, we show that, if $(v, u) \in \min(\mathcal{R}(\mathcal{T})\uparrow)$, then $|u| \leq \text{in}_{\max} = \sum_{i=1}^{|Q|} ((2|Q|)^i \cdot |\Sigma|)$ and $|v| \leq \text{out}_{\max} = \sum_{i=1}^{|Q|} ((2|Q|)^i \cdot |\Sigma| \cdot |Q| \cdot \gamma_{\max})$ where γ_{\max} represents the maximum length of an output on any transition in \mathcal{T} . To show this result, we need to define normalized runs as follows: A run is *normalized* if it visits each state at most once on each position x . In the following, we show that (v, u) can be generated by a normalized run ρ . Assume that (v, u) is accepted by a run ρ' which is not normalized. Then, we will have, in ρ' , two visits to some position x of the word in the same state q . After the first visit to position x in state q , the transducer has explored some positions till its

XX:12 On the Separability Problem of String Constraints

second visit to position x in state q . This part does not produce any output since (v, u) is a minimal word. We can delete this explored part of the run in between, obtaining again, an accepting run, which reads u while producing v . For example, in the figure if we have $q_6 = q_2$, then we have another run without visiting positions 1, 2 for a second time. Observe that repeating this procedure will lead to a normalized run ρ accepting (v, u) . The length of visiting sequences in a normalized run is $\leq |Q|$ and hence the number of visiting sequences is at most exponential in $|Q|$, precisely it is $\leq \sum_{i=1}^{|Q|} (2|Q|)^i$. Suppose $|u| > \text{in}_{\max}$. Then there exists a visiting sequence which is repeated on reading the same input symbol in the accepting run of u , at positions $i \neq j$. By deleting the part between the i th and $(j-1)$ th position, we again obtain an accepting run over a word u' , which is a strict subword of u , and whose output v' is also a subword (may not be strict) of v , a contradiction to $(v, u) \in \min(\mathcal{R}(\mathcal{T})\uparrow)$. Now suppose $|u| \leq \text{in}_{\max}$ but $|v| > \text{out}_{\max}$. We saw that in the normalized run, each visiting sequence has length at most $|Q|$. Then, since $|u| \leq \text{in}_{\max}$, on reading each position of u , at most $(|Q|)^{\gamma_{\max}}$ symbols can be produced. Hence, we have $|v| \leq (|Q|)^{\gamma_{\max}} \cdot |u|$. ◀

From Theorem 12 and Lemma 17, the following result holds:

► **Lemma 18.** *The 2-PosPTL separability problem for 2NFT is in EXPSpace.*

Proof. Using Theorem 12, we know that $\mathcal{R}(\mathcal{T}_1)\uparrow \cap \mathcal{R}(\mathcal{T}_2) = \emptyset$ iff \mathcal{T}_1 and \mathcal{T}_2 are 2-PosPTL separable. Here is an NEXPSpace algorithm.

(1) Guess some (v, u) s.t. the lengths of v, u are at most as given by the proof of Lemma 17.
(2) Check if $(v, u) \in \mathcal{R}(\mathcal{T}_1)$. If yes, then do (3). Else exit.
(3) Check if $(v, u) \uparrow \cap \mathcal{R}(\mathcal{T}_2) \neq \emptyset$. The guessed word (v, u) has exponential length in the size of \mathcal{T}_1 . To check if $(v, u) \in \mathcal{R}(\mathcal{T}_1)$, we construct another transducer \mathcal{T}'_1 that first checks that its input word is u , then it comes back to \vdash and starts simulating \mathcal{T}_1 , while also keeping track, longer and longer prefixes of v . We then compare those prefixes with the output produced by \mathcal{T}_1 . This gives rise to exponentially many states (maintaining prefixes of u and v) and we finish when \mathcal{T}_1 enters an accepting state, and at the same time, the produced word is v . Since $\mathcal{R}(\mathcal{T}'_1) = \{(v, u)\} \cap \mathcal{R}(\mathcal{T}_1)$ by construction, checking if $(v, u) \in \mathcal{R}(\mathcal{T}_1)$ can be reduced to the emptiness problem of \mathcal{T}'_1 . After this, we check the emptiness of $(v, u) \uparrow \cap \mathcal{R}(\mathcal{T}_2)$. This is done as follows. First, construct automata $\mathcal{A}_u, \mathcal{A}_v$ accepting languages $\{u\}\uparrow$ and $\{v\}\uparrow$ respectively. The number of states of $\mathcal{A}_u, \mathcal{A}_v$ are exponential in the number of states of \mathcal{T}_1 , since the lengths of u, v have this bound. Then, we construct a transducer \mathcal{T}'_2 such that $\mathcal{R}(\mathcal{T}'_2) = \{(v, u)\} \uparrow \cap \mathcal{R}(\mathcal{T}_2)$ in a similar manner as \mathcal{T}'_1 . \mathcal{T}'_2 reads the input word while simulating \mathcal{A}_u . On entering an accepting state of \mathcal{A}_u , it comes back to \vdash . Then it simulates \mathcal{T}_2 , and, on the outputs produced, simulates \mathcal{A}_v . If \mathcal{A}_v enters an accepting state at the same time \mathcal{T}_2 accepts, then we are done. The state space of \mathcal{T}'_2 is exponential in the states of \mathcal{T}_1 and linear in the states of \mathcal{T}_2 . Since $\mathcal{R}(\mathcal{T}'_2) = \{(v, u)\} \uparrow \cap \mathcal{R}(\mathcal{T}_2)$, checking the emptiness of $(v, u) \uparrow \cap \mathcal{R}(\mathcal{T}_2)$ can be reduced to checking the emptiness problem of \mathcal{T}'_2 . The emptiness problem for 2NFT is known to be PSPACE-COMplete [46]. Thus, in our case, the emptiness of \mathcal{T}'_1 and \mathcal{T}'_2 can be achieved in space exponential in \mathcal{T}_1 . Since we can handle the second and third steps in exponential space, we obtain an NEXPSpace algorithm. By Savitch's Theorem, we obtain the EXPSpace complexity. ◀

From Right-sided SL formulas to 2NFT. Hereafter, we show how to encode the set of solutions of a right-sided SL formula using 2NFT. Let Σ be an alphabet and $\# \notin \Sigma$.

► **Lemma 19.** *Let Ψ be a right-sided SL formula over Σ , with x_1, x_2, \dots, x_n as its set of variables. Then, it is possible to construct, in polynomial time, a 2NFT \mathcal{A}_Ψ such that*

$\mathcal{R}(\mathcal{A}_\Psi) = \{(u_1 \# u_2 \# \dots \# u_n, w_1 \# w_2 \# \dots \# w_n) \mid u_1 \# u_2 \# \dots \# u_n \in \text{Encode}(\mathcal{L}(\Psi)) \text{ and } w_i = u_i \text{ if } x_i \text{ is an independent variable}\}.$

Proof. Let us assume that Ψ is of the form $\bigwedge_{i=1}^n y_i \in \mathcal{L}(\mathcal{A}_i) \wedge \bigwedge_{i=1}^k (y_i, t_i) \in \mathcal{R}(\mathcal{T}_i)$ with y_1, \dots, y_n is a permutation of x_1, \dots, x_n . Let $\pi : [1, n] \rightarrow [1, n]$ be the mapping that associates to each index $i \in [1, n]$, the index $j \in [1, n]$ s.t. $x_i = y_j$ (or $x_i = y_{\pi(i)}$). We construct \mathcal{A}_Ψ as follows: \mathcal{A}_Ψ reads n words over Σ separated by $\#$ as input. We explain hereafter the working of \mathcal{A}_Ψ when it produces the assignment for x_1 (the other variables are handled in similar manner).

- Assume that x_1 is a dependent variable. Let $\varphi_{\pi(1)} = (y_{\pi(1)}, t_{\pi(1)}) \in \mathcal{R}(\mathcal{T}_{\pi(1)})$, with $t_{\pi(1)} = x_{i_1} x_{i_2} \dots x_{i_c}$ and $x_{i_j} \in \{y_{k+1}, y_{k+2}, \dots, y_n\}$ for all j . First, \mathcal{A}_Ψ reads x_{i_1} i.e. the first variable in $t_{\pi(1)}$. To read x_{i_1} , it skips $(i_1 - 1)$ many blocks separated by $\#$ s of the input, and comes to w_{i_1} . On the first symbol of w_{i_1} , \mathcal{A}_Ψ starts mimicking transitions of $\mathcal{T}_{\pi(1)}$ from its initial state, while producing the same output as $\mathcal{T}_{\pi(1)}$. On the same output, \mathcal{A}_Ψ mimics the transitions of $\mathcal{A}_{\pi(1)}$ starting from the initial state to check the membership constraint of $y_{\pi(1)}$. This can be done by a product construction between $\mathcal{A}_{\pi(1)}$ and $\mathcal{T}_{\pi(1)}$. For instance, \mathcal{A} will have a transition $((p, q), a, (p', q'), b, 1)$ (resp. $((p, q), a, (p', q'), b, 0)$), if there are transitions $(p, (b, a), p')$ (resp. $(p, (b, \epsilon), p')$) in $\mathcal{T}_{\pi(1)}$ and (q, b, q') in $\mathcal{A}_{\pi(1)}$. If it reaches $\#$ or \neg in the input, it remembers the current states of $\mathcal{T}_{\pi(1)}$ and $\mathcal{A}_{\pi(1)}$, say (p_1, q_1) in its control state. Next, \mathcal{A}_Ψ reads x_{i_2} in the input. To read x_{i_2} , \mathcal{A}_Ψ moves to \vdash and then changes direction. As before it reaches x_{i_2} by skipping $(i_2 - 1)$ many $\#$ s, and starts reading the input (the first symbol of w_{i_2}) from the state (p_1, q_1) stored in the finite control. Transitions are similar to explained above. This procedure is repeated to read $x_{i_3} \dots x_{i_c}$. After reading x_{i_c} , if the next state contains the pair (p_c, q_c) , where p_c (resp. q_c) is a final state of $\mathcal{T}_{\pi(1)}$ (resp. $\mathcal{A}_{\pi(1)}$), we can say that the output produced till now satisfies $\varphi_{\pi(1)}$ and $y_{\pi(1)} \in \mathcal{L}(\mathcal{A}_{\pi(1)})$.
- Assume now that x_1 is an independent variable, then \mathcal{A}_Ψ needs to read x_1 . We need a single pass of the input which verifies if the first block corresponding to value of x_1 in input indeed satisfies its corresponding membership constraint. During this pass \mathcal{A}_Ψ mimics transitions of $\mathcal{A}_{\pi(1)}$ starting from its initial states, and outputs the same letter as input.

The above procedure is repeated for all variables from x_2 to x_n . After each pass, \mathcal{A} moves to \vdash and then changes direction. Irrespective of whether x_i is dependent or not, while going from x_i to x_{i+1} , $i \in [1, n - 1]$, \mathcal{A} outputs a $\#$ as separator. From the description above, it can be seen that if x_i is independent, then its evaluation u_i given as the i th block of the input is equal to the output w_i , and if x_i is a dependent variable, then the output block w_i is the output of $\mathcal{T}_{\pi(i)}$. \blacktriangleleft

Notice that the above construction of 2NFT relies on the right-sidedness: if a variable x_i appears in the output of \mathcal{T}_i and also in the input of \mathcal{T}_k for some k , then we will have to store the produced evaluation of x_i in order to use it later on when processing \mathcal{T}_k . However, there is no way to store the produced evaluation of x_i or compare it with its input evaluation. Next, we show that the PosPTL separability problem for right-sided formulas can be reduced to its corresponding problem for 2NFT.

► **Lemma 20.** *Let Ψ_1 and Ψ_2 be two right-sided SL formula, with x_1, \dots, x_n as their set of variables. Let \mathcal{A}_{Ψ_1} and \mathcal{A}_{Ψ_2} be the two 2NFTs encoding, respectively, the set of solutions of Ψ_1 and Ψ_2 (as described in Lemma 19). Then, the two formulae Ψ_1 and Ψ_2 are separable by n -PosPTL iff $\mathcal{R}(\mathcal{A}_1)$ and $\mathcal{R}(\mathcal{A}_2)$ are separable by a 2-PosPTL.*

Proof. Let $\mathcal{R}(\mathcal{A}_{\Psi_1})$ and $\mathcal{R}(\mathcal{A}_{\Psi_2})$ be separable by a 2-PosPTL L . By definition, L is a Boolean combination (except complementation) of piece languages of words over the two

tuple alphabet $(\Sigma \cup \{\#\})^2$. We can assume w.l.o.g. that L is the union of piece languages. This is possible since the intersection of two piece languages can be rewritten as a union of piece languages. Consider $L' = L \cap (R \times R)$, where R is a regular language consisting of words having exactly $(n-1)$ $\#$ s. We claim that L' can be rewritten as the union of languages of the form $[L_1 \# L_2 \# \dots \# L_n] \times [R_1 \# R_2 \# \dots \# R_n]$ where the L_i s and R_i s are piece languages over Σ , and that L' is also a separator of $\mathcal{R}(\mathcal{A}_{\Psi_1})$ and $\mathcal{R}(\mathcal{A}_{\Psi_2})$.

We prove this claim inductively. As a base case consider L to be a piece language $((\Sigma \cup \{\#\})^*)^2(a_1, b_1)((\Sigma \cup \{\#\})^*)^2 \dots (a_m, b_m)((\Sigma \cup \{\#\})^*)^2$. Let S be a finite set containing only the *minimal* words of the form (w, w') such that $a_1 a_2 \dots a_m \preceq w$, $b_1 b_2 \dots b_m \preceq w'$, and the symbol $\#$ appears exactly $(n-1)$ -times in w and w' . Thus $L \cap (R \times R) = \bigcup_{(a'_1 \dots a'_k, b'_1 \dots b'_\ell) \in S} [\Sigma^* a'_1 \Sigma^* a'_2 \dots a'_k \Sigma^*] \times [\Sigma^* b'_1 \Sigma^* b'_2 \dots b'_\ell \Sigma^*]$. So $L \cap (R \times R)$ is the union of piece languages of the form $[L_1 \# L_2 \# \dots \# L_n] \times [R_1 \# R_2 \# \dots \# R_n]$ where the L_i s and R_i s are piece languages over Σ . Now assume that L is of the form $L_1 \cup L_2$. It is easy to see that $L \cap (R \times R)$ is equivalent to $(L_1 \cap (R \times R)) \cup (L_2 \cap (R \times R))$. Thus we can use our induction hypothesis to show that $L \cap (R \times R)$ is the union of languages of the form $[L_1 \# L_2 \# \dots \# L_n] \times [R_1 \# R_2 \# \dots \# R_n]$ where L_i and R_i s are piece languages over Σ . Next we prove that L' is a separator of $\mathcal{R}(\mathcal{A}_{\Psi_1})$ and $\mathcal{R}(\mathcal{A}_{\Psi_2})$. Indeed if $(v, u) \in \mathcal{R}(\mathcal{A}_{\Psi_1})$, then $(v, u) \in R \times R$, by definition of $\mathcal{R}(\mathcal{A}_{\Psi_1})$. Since L is a separator, we have $(v, u) \in L$ and hence $(v, u) \in L'$. Suppose $(v, u) \in \mathcal{R}(\mathcal{A}_{\Psi_2}) \cap L'$, then $(v, u) \in L \cap \mathcal{R}(\mathcal{A}_{\Psi_1})$ since $(v, u) \in L'$, and $L' \subseteq L$, which is a contradiction with the assumption that L is a separator.

Now we are in a condition to provide n -PosPTL separator for $\mathcal{L}(\Psi_1)$ and $\mathcal{L}(\Psi_2)$, using L' . Given a language of the form $[L_1 \# L_2 \# \dots \# L_n] \times [R_1 \# R_2 \# \dots \# R_n]$ where L_i and R_i s are piece languages, we associate to it an n -PosPTL equivalent to $((L_1 \cap R_1) \times (L_2 \cap R_2) \times \dots \times (L_n \cap R_n))$: the idea is to generate the n dimensions in the n -PosPTL from the n $\#$ -separated blocks in two dimensions. This definition is extended in the straightforward manner to union of piece languages. Let K be the n -PosPTL associated to L' . K is indeed a separator of $\mathcal{L}(\Psi_1)$ and $\mathcal{L}(\Psi_2)$: Suppose $\mathbf{v} = (w_1, \dots, w_n) \in \mathcal{L}(\Psi_1)$, then $(w_1 \# \dots \# w_n, w_1 \# \dots \# w_n) \in \mathcal{R}(\mathcal{A}_{\Psi_1})$ (from the definition of \mathcal{A}_{Ψ_1}). Since L' is a separator, $(w_1 \# \dots \# w_n, w_1 \# \dots \# w_n) \in L'$. By construction of K , $(w_1, \dots, w_n) \in K$. Assume $\mathbf{v} = (w_1, \dots, w_n) \in \mathcal{L}(\Psi_2) \cap K$, then $(w_1 \# \dots \# w_n, w_1 \# \dots \# w_n) \in L'$. Since $L' \cap \mathcal{R}(\mathcal{A}_{\Psi_2}) = \emptyset$, then $(w_1 \# \dots \# w_n, w_1 \# \dots \# w_n) \notin \mathcal{R}(\mathcal{A}_{\Psi_2})$. By definition of \mathcal{A}_{Ψ_2} , if $(w_1, \dots, w_n) \in \mathcal{L}(\Psi_2)$, then $(w_1 \# \dots \# w_n, w_1 \# \dots \# w_n) \in \mathcal{R}(\mathcal{A}_{\Psi_2})$. Hence contradiction.

For the other direction of the proof, assume the n -PosPTL S is a separator of $\mathcal{L}(\Psi_1)$ and $\mathcal{L}(\Psi_2)$. Then S can be rewritten as the union of $(L_1 \times L_2 \times \dots \times L_n)$ where L_i s are piece languages. Replace each n -piece language $(L_1 \times L_2 \times \dots \times L_n)$ of S with the 2-piece language $(L'_1 \# L'_2 \# \dots \# L'_n) \times ((\Sigma \cup \{\#\})^* \# \dots \# (\Sigma \cup \{\#\})^*)$, where $L'_1 = (\Sigma \cup \{\#\})^* a_1 (\Sigma \cup \{\#\})^* \dots a_n (\Sigma \cup \{\#\})^*$ if $L_1 = \Sigma^* a_1 \Sigma^* \dots a_n \Sigma^*$. Denote the union of such languages by S' . It is a 2-PosPTL over $(\Sigma \cup \{\#\})$. We show that S' is a 2-PosPTL separator of $\mathcal{R}(\mathcal{A}_{\Psi_1})$ and $\mathcal{R}(\mathcal{A}_{\Psi_2})$. Let $(\mathbf{v}, \mathbf{u}) = (v_1 \# \dots \# v_n, u_1 \# \dots \# u_n) \in \mathcal{R}(\mathcal{A}_{\Psi_1})$, then $(v_1, \dots, v_n) \in \mathcal{L}(\Psi_1)$, and thus $(v_1, \dots, v_n) \in S$ (since S is a separator). This implies that $(\mathbf{v}, \mathbf{u}) \in S'$ by its construction. Suppose $(\mathbf{v}, \mathbf{u}) = (v_1 \# \dots \# v_n, u_1 \# \dots \# u_n) \in \mathcal{R}(\mathcal{A}_{\Psi_2}) \cap S'$, then $(v_1, v_2, \dots, v_n) \in \mathcal{L}(\Psi_2)$. Also $(v_1, v_2, \dots, v_n) \in S$ by construction of S' . This leads to a contradiction that S is separator of $\mathcal{L}(\Psi_1)$ and $\mathcal{L}(\Psi_2)$. So S' is a 2-PosPTL separator of $\mathcal{R}(\mathcal{A}_{\Psi_1})$ and $\mathcal{R}(\mathcal{A}_{\Psi_2})$. ◀

► **Theorem 21.** *The n -PosPTL separability of right-sided SL formulas is in EXPSpace and is PSPACE-HARD.*

Proof. Given two right-sided SL formulas Ψ_1 and Ψ_2 , one can construct corresponding two way transducers \mathcal{A}_{Ψ_1} and \mathcal{A}_{Ψ_2} with polynomial states, as mentioned in Lemma 19.

Thanks to Lemma 20, the n -PosPTL separability reduces to 2-PosPTL separability of $\mathcal{R}(\mathcal{A}_{\Psi_1})$ and $\mathcal{R}(\mathcal{A}_{\Psi_2})$. The 2-PosPTL separability of 2NFTs is in EXPSPACE (Lemma 18). Hence n -PosPTL separability of SL formulae is also in EXPSPACE. For the PSPACE-HARD lower bound, we reduce the emptiness of k -NFA intersection to PosPTL separability of right sided SL. Let $\mathcal{A}_1, \dots, \mathcal{A}_k$ be k -NFA. We want to decide if $\bigcap_{i=1}^k \mathcal{A}_i = \emptyset$. Let Ψ_1 be $\bigwedge_{i=1}^k x_i = x \wedge \bigwedge_{i=1}^k (x_i \in \mathcal{A}_i)$, and Ψ_2 be $x \in \Sigma^*$. Ψ_1 and Ψ_2 are PosPTL separable iff $\bigcap_{i=1}^k \mathcal{A}_i = \emptyset$. ◀

6 Examples

We conclude with 2 examples. One, we give a string program whose safety checking boils down to checking the separability of two SL string constraints. Second, we illustrate how to compute the PosPTL separator given two SL string constraints, using the 2NFT encoding and Theorem 12.

► **Example 22.** As a practical motivation of PosPTL (and PTL), consider the following pseudo-PHP code obtained as a variation of the code at [1]. In this code, a user is prompted to change his password by entering the new password twice.

```
str old = real_escape_string(oldIn);
str new1 = real_escape_string(newIn1);
str new2 = real_escape_string(newIn2);
str pass = database_query("SELECT password FROM users WHERE userID=" userID);
if (old == pass AND new1 == new2 AND new1 != old )
    if (newIn1==newIn2 AND newIn1 != oldIn)
        str query = "UPDATE users SET password=" new1 "WHERE userID=" userID;
        database_query(query);
```

The user inputs the old password `oldIn` and the new password twice : `newIn1` and `newIn2`. These are sanitized and assigned to `old`, `new1` and `new2` respectively. The old sanitized password is compared with the value `pass` from the database to authenticate the user, and also with the new sanitized password to check that a different password has been chosen, and finally, the sanitized new passwords entered twice are checked to be the same. Sanitization ensures that there are no SQL injections. To ensure the absence of SQL attacks, we require that the query `query` does not belong to a regular language Bad of bad patterns over some finite alphabet Σ (i.e., the program is safe). This safety condition can be expressed as the unsatisfiability of the following formula φ given by

$\text{new1} = T(\text{newIn1}) \wedge \text{new2} = T(\text{newIn2}) \wedge \text{old} = T(\text{oldIn}) \wedge \text{new1} = \text{new2} \wedge \text{pass} = \text{old} \wedge \text{old} \neq \text{new1} \wedge \text{newIn1} = \text{newIn2} \wedge \text{query} = u \cdot \text{new1} \cdot v \cdot \text{userID} \wedge \text{query} \in \text{Bad}.$

Note that the check $\text{new1} = \text{new2}$ has to be done by the server to ensure the sanitized new passwords entered twice are same; however, the check $\text{newIn1} = \text{newIn2}$ is not redundant, since it can happen that post sanitization, the passwords may agree, but not before. The sanitization on lines 1, 2 and 3 is represented by the transducer T and u, v are the constant strings from line 7. It is easy to see that the program given here is safe iff the formula φ is unsatisfiable. Observe that the formula φ is not in the straight line fragment [32] since variable `new1` has two assignments. Further, it also has a non-benign chain (see below) making it fall out of the fragment of string programs handled in [5]. However the formula φ can be rewritten as a conjunction of the two formula φ_1 and φ_2 in straight-line form where $\varphi_1 : \text{new1} = T(\text{newIn1}) \wedge \text{old} = T(\text{oldIn}) \wedge \text{pass} = \text{old} \wedge \text{query} = u \cdot \text{new1} \cdot v \cdot \text{userID} \wedge \text{query} \in \text{Bad}$ $\varphi_2 : \text{new2} = T(\text{newIn2}) \wedge \text{new1} = \text{new2} \wedge \text{old} \neq \text{new1} \wedge \text{newIn1} = \text{newIn2}.$

XX:16 On the Separability Problem of String Constraints

It is easy to see that the program is safe iff φ_1 and φ_2 are separable by the PosPTL language that associates to each string variable Σ^* s.

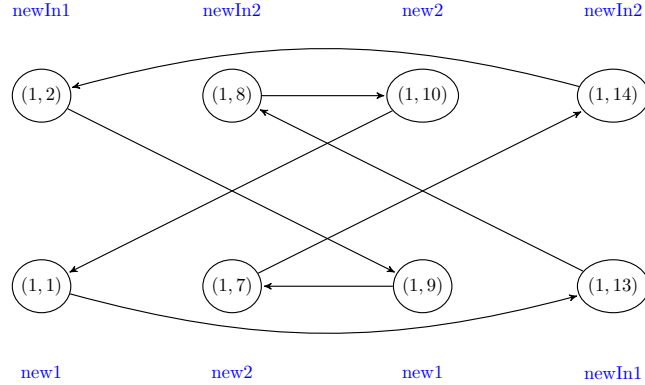
The string program falls out of the chain-free fragment

To define the chain-free fragment, [5] introduces the notion of a splitting graph. Assume we are given a string constraint $\Psi = \bigwedge_{i=1}^n \varphi_i$ having n relational constraints, each of the form $\varphi_j = R_j(t_{2j-1}, t_j)$. Let each term t_i be a concatenation of variables $x_{i,1} \dots x_{i,m_i}$. Given such a string constraint, its splitting graph contains nodes of the form $\{(i, j) \mid 1 \leq j \leq 2n, 1 \leq i \leq n_j\}$. Each node (i, j) is labeled by the variable $x_{j,i}$. The node $(i, 2j-1)$ (resp. $(i, 2j)$) represents the i th term in the left side (respectively, the right side) of constraint φ_j . There is an edge from node p to node q if there exists a node p' (different from q) such that p and p' represent the nodes corresponding to different sides of the same constraint (say φ_i) and p' and q have the same label. An edge (p, q) , with $p = (i, j)$ is labeled by the j th constraint φ_j .

A *chain* in the graph is a sequence of edges of the form $(p_0, p_1)(p_1, p_2) \dots (p_n, p_0)$. A chain is a *benign* chain if (1) all relational constraints corresponding to the edges are all of the form $R(x, t)$ where x is a single variable (t is a term as usual) and length preserving, and (2) the sequence of positions p_0, p_1, \dots, p_n all correspond to the left side (or all to the right side).

Recall from φ_1, φ_2 above, that we consider for the string program, the straight-line constraints

$\phi_1 : \text{new1} = T(\text{newIn1}), \phi_2 : \text{old} = T(\text{oldIn}), \phi_3 : \text{pass} = \text{old}, \phi_4 : \text{new2} = T(\text{newIn2}),$
 $\phi_5 : \text{new1} = \text{new2}, \phi_6 : \text{old} \neq \text{new1}, \phi_7 : \text{newIn1} = \text{newIn2}.$



■ **Figure 1** splitting graph for the example

By this definition, the part of the splitting graph which induces a non-benign chain is shown in figure 1: Observe that there is a chain formed from the nodes of splitting graph which is: $(1, 1) \rightarrow (1, 13) \rightarrow (1, 8) \rightarrow (1, 10) \rightarrow (1, 1)$. This chain is not benign since the transducer functions are not length preserving.

► **Example 23.** We illustrate how to compute the PosPTL separator given two SL string constraints, using the 2NFT encoding and the result from Theorem 12.

Consider two SL string constraints

$$\Psi_1 : (x, ay) \in \mathcal{R}(\mathcal{T}_1) \wedge y \in (e + f)^*, \Psi_2 : (y, x) \in \mathcal{R}(\mathcal{T}_2) \wedge x \in (e + f)^*$$

The alphabet is $\Sigma = \{a, e, f\}$. The transducer \mathcal{T}_1 and \mathcal{T}_2 implement the following functions $\mathcal{T}_1(w) = w\uparrow$ for $w \in a(e + f)^*$, and $\mathcal{T}_2(w') = w'\downarrow$ for $w' \in (e + f)^*$.

Note that $\Psi_1 \wedge \Psi_2$ is not in SL, also it has a non-benign chain as defined in [5]. We show that the languages of formulas Ψ_1 and Ψ_2 are separable by a PosPTL separator, thereby concluding that $\Psi_1 \wedge \Psi_2$ is unsatisfiable.

To decide the separability and get this separator, we follow the procedure of Theorem 12.

- Encode the solutions of formulae to 2NFT.
- $\mathcal{R}(\mathcal{A}_{\Psi_1}) = \{(v\#w, u\#w) \mid u \text{ is arbitrary}, w \in (e+f)^*, \text{ and } v \in \{aw\}^\uparrow\}$.
- $\mathcal{R}(\mathcal{A}_{\Psi_2}) = \{(w\#v, w\#u) \mid u \text{ is arbitrary}, w \in (e+f)^*, \text{ and } v \in \{w\}^\downarrow\}$.
- Each component in $\mathcal{R}(\mathcal{A}_{\Psi_1}), \mathcal{R}(\mathcal{A}_{\Psi_2})$ is of the form $\eta(x)\#\eta(y)$.
- Decide the emptiness of $\mathcal{R}(\mathcal{A}_{\Psi_1})^\uparrow \cap \mathcal{R}(\mathcal{A}_{\Psi_2})$. This check reduces to the emptiness of $\min(\mathcal{R}(\mathcal{A}_{\Psi_1})^\uparrow \cap \mathcal{R}(\mathcal{A}_{\Psi_2}))$. The set of minimal words of $\mathcal{R}(\mathcal{A}_{\Psi_1})^\uparrow$ is $M = \{(a\#, \#)\}$ and $M^\uparrow \cap \mathcal{R}(\mathcal{A}_{\Psi_2})$ is clearly empty.

Hence $M^\uparrow = ((\Sigma \cup \{\#\})^* a (\Sigma \cup \{\#\})^* \# (\Sigma \cup \{\#\})^*) \times ((\Sigma \cup \{\#\})^* \# (\Sigma \cup \{\#\})^*)$ is a PosPTL separator of $\mathcal{R}(\mathcal{A}_{\Psi_1}), \mathcal{R}(\mathcal{A}_{\Psi_2})$. This gives the PosPTL separator $\Sigma^* a \Sigma^* \times \Sigma^*$ for Ψ_1, Ψ_2 .

References

- 1 PHP Tutorials (2019). URL: <https://makephpsites.com/php-tutorials/user-management-tools/changing-passwords.php>.
- 2 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukás Holík, Ahmed Rezine, and Philipp Rümmer. Flatten and conquer: a framework for efficient analysis of string constraints. In *PLDI*. ACM, 2017.
- 3 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukás Holík, Ahmed Rezine, and Philipp Rümmer. Trau: SMT solver for string constraints. In *FMCAD*. IEEE, 2018.
- 4 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukás Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. String constraints for verification. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 150–166. Springer, 2014.
- 5 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bui Phi Diep, Lukás Holík, and Petr Janku. Chain-free string constraints. In Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza, editors, *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings*, volume 11781 of *Lecture Notes in Computer Science*, pages 277–293. Springer, 2019.
- 6 Parosh Aziz Abdulla and Richard Mayr. Priced timed petri nets. *Logical Methods in Computer Science*, 9(4), 2013. URL: [https://doi.org/10.2168/LMCS-9\(4:10\)2013](https://doi.org/10.2168/LMCS-9(4:10)2013), doi: 10.2168/LMCS-9(4:10)2013.
- 7 Mohamed Faouzi Atig, Benedikt Bollig, and Peter Habermehl. Emptiness of Ordered Multi-Pushdown Automata is 2ETIME-Complete. *Int. J. Found. Comput. Sci.*, 28(8):945–976, 2017.
- 8 Félix Baschenis, Olivier Gauwin, Anca Muscholl, and Gabriele Puppis. Minimizing resources of sweeping and streaming string transducers. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, volume 55 of *LIPICs*, pages 114:1–114:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.
- 9 Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. Z3str3: A string solver with theory-aware heuristics. In *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pages 55–59. IEEE, 2017.
- 10 Murphy Berzish, Yunhui Zheng, and Vijay Ganesh. Z3str3: A string solver with theory-aware branching. *CoRR*, abs/1704.07935, 2017.
- 11 Luca Breveglieri, Alessandra Cherubini, Claudio Citrini, and Stefano Crespi-Reghizzi. Multi-push-down languages and grammars. *Int. J. Found. Comput. Sci.*, 7(3):253–292, 1996. URL: <https://doi.org/10.1142/S0129054196000191>, doi: 10.1142/S0129054196000191.

- 12 Taolue Chen, Yan Chen, Matthew Hague, Anthony W. Lin, and Zhilin Wu. What is decidable about string constraints with the replaceall function. *PACMPL*, 2(POPL):3:1–3:29, 2018. URL: <https://doi.org/10.1145/3158091>, doi:10.1145/3158091.
- 13 Taolue Chen, Matthew Hague, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. URL: <http://doi.acm.org/10.1145/3290362>, doi:10.1145/3290362.
- 14 Taolue Chen, Matthew Hague, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. Decision procedures for path feasibility of string-manipulating programs with complex operations. *PACMPL*, 3(POPL):49:1–49:30, 2019.
- 15 Lorenzo Clemente, Wojciech Czerwinski, Slawomir Lasota, and Charles Paperman. Regular separability of parikh automata. In *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, volume 80 of *LIPIcs*, pages 117:1–117:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.
- 16 Bruno Courcelle. On constructing obstruction sets of words. *Bulletin of the EATCS*, 44:178–186, 1991.
- 17 Wojciech Czerwinski and Slawomir Lasota. Regular separability of one counter automata. *Logical Methods in Computer Science*, 15(2), 2019. URL: <https://lmcs.episciences.org/5563>.
- 18 Wojciech Czerwinski, Slawomir Lasota, Roland Meyer, Sebastian Muskalla, K. Narayan Kumar, and Prakash Saivasan. Regular separability of well-structured transition systems. In *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China*, volume 118 of *LIPIcs*, pages 35:1–35:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.
- 19 Wojciech Czerwinski, Wim Martens, and Tomás Masopust. Efficient separability of regular languages by subsequences and suffixes. In *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II*, volume 7966 of *Lecture Notes in Computer Science*, pages 150–161. Springer, 2013.
- 20 Wojciech Czerwinski, Wim Martens, Lrijn van Rooijen, Marc Zeitoun, and Georg Zetsche. A characterization for decidable separability by piecewise testable languages. *Discrete Mathematics & Theoretical Computer Science*, 19(4), 2017.
- 21 Vijay Ganesh and Murphy Berzish. Undecidability of a theory of strings, linear arithmetic over length, and string-number conversion. *CoRR*, abs/1605.09442, 2016. URL: <http://arxiv.org/abs/1605.09442>, arXiv:1605.09442.
- 22 Vijay Ganesh, Mia Minnes, Armando Solar-Lezama, and Martin C. Rinard. Word equations with length constraints: What’s decidable? In *Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers*, volume 7857 of *Lecture Notes in Computer Science*, pages 209–226. Springer, 2012.
- 23 Vijay Ganesh, Mia Minnes, Armando Solar-Lezama, and Martin C. Rinard. (un)decidability results for word equations with length and regular expression constraints. *CoRR*, abs/1306.6054, 2013. URL: <http://arxiv.org/abs/1306.6054>, arXiv:1306.6054.
- 24 Sheila Ginsburg, Seymour; Greibach. Abstract families of languages. In *Eighth Annual Symposium on Switching and Automata Theory*. IEEE, 1967.
- 25 G. Higman. Ordering by divisibility in abstract algebras. *Proc. London Math. Soc.* (3), 2(7), 1952.
- 26 Lukás Holík, Petr Janku, Anthony W. Lin, Philipp Rümmer, and Tomás Vojnar. String constraints with concatenation and transducers solved efficiently. *PACMPL*, 2(POPL):4:1–4:32, 2018. URL: <https://doi.org/10.1145/3158092>, doi:10.1145/3158092.
- 27 John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

- 28 Scott Kausler and Elena Sherman. Evaluation of string constraint solvers in the context of symbolic execution. In *ASE '14*. ACM, 2014.
- 29 Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: a solver for string constraints. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*, pages 105–116. ACM, 2009.
- 30 Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In *CAV'14*, volume 8559 of *LNCS*. Springer, 2014.
- 31 Tianyi Liang, Andrew Reynolds, Nestan Tsiskaridze, Cesare Tinelli, Clark W. Barrett, and Morgan Deters. An efficient SMT solver for string constraints. *Formal Methods in System Design*, 48(3):206–234, 2016. URL: <https://doi.org/10.1007/s10703-016-0247-6>, doi: 10.1007/s10703-016-0247-6.
- 32 Anthony Widjaja Lin and Pablo Barceló. String solving with word equations and transducers: towards a logic for analysing mutation XSS. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 123–136. ACM, 2016.
- 33 Kenneth L. McMillan. Interpolation and sat-based model checking. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.
- 34 Kenneth L. McMillan. An interpolating theorem prover. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2004.
- 35 Kenneth L. McMillan. Lazy abstraction with interpolants. In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.
- 36 Christophe Morvan. On rational graphs. In *Foundations of Software Science and Computation Structures*, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.
- 37 Thomas Place, Lorijn van Rooijen, and Marc Zeitoun. Separating regular languages by piecewise testable and unambiguous languages. In *Mathematical Foundations of Computer Science 2013 - 38th International Symposium, MFCS 2013, Klosterneuburg, Austria, August 26-30, 2013. Proceedings*, volume 8087 of *Lecture Notes in Computer Science*, pages 729–740. Springer, 2013.
- 38 Wojciech Plandowski. An efficient algorithm for solving word equations. In *Proceedings of the Thirty-eighth Annual ACM Symposium on Theory of Computing, STOC '06*, pages 467–476, New York, NY, USA, 2006. ACM. URL: <http://doi.acm.org/10.1145/1132516.1132584>.
- 39 Andrew Reynolds, Maverick Woo, Clark W. Barrett, David Brumley, Tianyi Liang, and Cesare Tinelli. Scaling up DPLL(T) string solvers using context-dependent simplification. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 453–474. Springer, 2017.
- 40 Arto Salomaa. Formal languages. In *Formal Languages*. Academic Press, 1987.
- 41 Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berkeley/Oakland, California, USA*, pages 513–528. IEEE Computer Society, 2010.
- 42 Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. FLAX: systematic discovery of client-side validation vulnerabilities in rich web applications. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*. The Internet Society, 2010.

- 43 Imre Simon. Piecewise testable events. In *Automata Theory and Formal Languages, 2nd GI Conference, Kaiserslautern, May 20-23, 1975*, volume 33 of *Lecture Notes in Computer Science*, pages 214–222. Springer, 1975.
- 44 Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *CCS'14*. ACM, 2014.
- 45 Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. Progressive reasoning over recursively-defined strings. In *CAV'16*, volume 9779 of *LNCS*. Springer, 2016.
- 46 Moshe Y. Vardi. A note on the reduction of two-way automata to one-way automata. *Inf. Process. Lett.*, 30(5):261–264, 1989. URL: [https://doi.org/10.1016/0020-0190\(89\)90205-6](https://doi.org/10.1016/0020-0190(89)90205-6), doi:10.1016/0020-0190(89)90205-6.
- 47 Hung-En Wang, Tzung-Lin Tsai, Chun-Han Lin, Fang Yu, and Jie-Hong R. Jiang. String analysis via automata manipulation with logic circuit representation. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, volume 9779 of *Lecture Notes in Computer Science*, pages 241–260. Springer, 2016.
- 48 Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Stranger: An automata-based string analysis tool for PHP. In *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*, pages 154–157. Springer, 2010.
- 49 Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: A Z3-based string solver for web application analysis. In *ESEC/FSE'13*. ACM, 2013.

Appendix

A Proof from Section 2 (Proof of lemma 1)

Given a string constraint Ψ over variables x_1, \dots, x_m , one can construct two SL string constraints Ψ_1 and Ψ_2 s.t. Ψ and $\Psi_1 \wedge \Psi_2$ are equisatisfiable.

By definition Ψ is a conjunction of atomic regular membership and relational string constraints i.e. $\Psi = \bigwedge_{i=1}^n \phi_i \wedge \bigwedge_{i=1}^k \varphi_i$ where ϕ_i s are membership constraints and φ_i s are relational constraints. Since regular languages are closed under intersection, one can combine the constraints of the form $x_i \in L_1$ and $x_i \in L_2$ as $x_i \in L$ where $L = L_1 \cap L_2$. This way we have at most one membership constraint for each variable. We can add the conjunction of membership constraints to either Ψ_1 or Ψ_2 or both.

Next, we show that by introducing new variables corresponding to each relational constraint φ , we can partition Ψ into two SLs. For each relational constraint $\varphi : (x_i, t_i) \in \mathcal{R}(\mathcal{T})$, we add a new variable u_φ and add a formula $u_\varphi = x_i$ (or $(u_\varphi, x_i) \in \text{Id}$, where Id is the identity) to Ψ_1 and $(u_\varphi, t_i) \in \mathcal{R}(\mathcal{T})$ to Ψ_2 . After iterating this procedure for all relational constraints φ , we claim that we have SL formulas in Ψ_1 as well as Ψ_2 .

Observe that in Ψ_i for $i = 1, 2$, variables x_1, \dots, x_m appear only as part of (i) membership constraint or (ii) input to relational constraint. These variables are independent. Newly introduced variables u_{φ_j} appear as only output of exactly one relational constraint in Ψ_1 and Ψ_2 . Hence the variables can be ordered as u_{φ_j} s having lower precedence than x_i s, and both formulas Ψ_1 and Ψ_2 preserve the SL syntax. Clearly Ψ and $\Psi_1 \wedge \Psi_2$ are equisatisfiable: If we have a solution for x_i s in Ψ , then the same assignment works in $\Psi_1 \wedge \Psi_2$, in addition u_{φ_j} s are assigned the values based on values of x_i s. On other side, given a solution for $\Psi_1 \wedge \Psi_2$, the same assignment of variables x_i s works for Ψ .

B Proof from Section 3 (Proof of theorem 2)

In the following, we show that the REG separability problem is undecidable even for the subclass SL of string constraints.

Proof. The proof is done by reduction from the halting problem of Turing machines. Consider a deterministic Turing machine M whose set of states is Q and tape alphabet is Σ , with $a, \#, \$ \notin \Sigma$. Encode a configuration w of M as a word in $\Sigma^* \times (Q \times \Sigma) \times \Sigma^*$ and use $w \rightarrow_M w'$ to denote that M can reach the configuration w' from the configuration w in one step. Let w_1 denote the initial configuration of M . It can be easily seen that one can design a transducer that accepts the language $\{(w, w') \mid w \rightarrow_M w'\}$. This construction can be extended to design two transducers \mathcal{T}_1 and \mathcal{T}_2 accepting respectively the following two languages:

$$L_1 = \{(w_1 \# w_3 \# \dots \# w_{2k-1} \$ a^j, w_2 \# w_4 \# \dots \# w_{2k} \$ w'_1 \# \dots \# w'_j \#) \mid k \geq 1, w_{2i-1} \rightarrow_M w_{2i} \text{ for } i \in [1, k] \text{ and } w'_1, \dots, w'_j \in \Sigma^* \times (Q \times \Sigma) \times \Sigma^* \text{ are arbitrary}\}$$

$$L_2 = \{(w_1 \# w_3 \# \dots \# w_{2k-1} \$ a^{2j}, w_2 \# w_4 \# \dots \# w_{2k} \$ w'_1 \# \dots \# w'_j \#) \mid k \geq 1, w_{2i-2} = w_{2i-1} \text{ for } i \in [2, k] \text{ and } w'_1, \dots, w'_j \in \Sigma^* \times (Q \times \Sigma) \times \Sigma^* \text{ are arbitrary}\}$$

Consider the SL formulae $\Psi_1 = (x, y \$ y) \in \mathcal{R}(\mathcal{T}_1)$ and $\Psi_2 = (x, y \$ y) \in \mathcal{R}(\mathcal{T}_2)$. Then, it is easy to see that

$$\mathcal{L}(\Psi_1) = \{(w_1 \# w_3 \# \dots \# w_{2k-1} \$ a^k, w_2 \# w_4 \# \dots \# w_{2k} \#) \mid k \geq 1, w_2, w_3, \dots, w_{2k} \in \Sigma^* \times (Q \times \Sigma) \times \Sigma^*, \text{ and } w_{2i-1} \rightarrow_M w_{2i} \text{ for } i \in [1, k]\}$$

XX:22 On the Separability Problem of String Constraints

$$\mathcal{L}(\Psi_2) = \{(w_1 \# w_3 \# \dots \# w_{2k-1} \$ a^{2k}, w_2 \# w_4 \# \dots \# w_{2k} \#) \mid k \geq 1, w_2, w_3, \dots, w_{2k} \in \Sigma^* \times (Q \times \Sigma) \times \Sigma^*, w_{2i-2} = w_{2i-1} \text{ for } i \in [2, k]\}$$

We prove that Ψ_1 and Ψ_2 are REG separable if and only if M halts.

Case 1. Suppose M does not halt. Assume that there exists a REG separator $\bigcup_{i=1}^{\ell} L_i \times L'_i$ separating languages of Ψ_1 and Ψ_2 . Let A_i, B_i respectively be DFA s.t. $L_i = \mathcal{L}(A_i), L'_i = \mathcal{L}(B_i)$. Let $n-1$ be the total number of states in A_1, \dots, A_{ℓ} . Consider the 2-tuple

$$(v_1, v_2) = (w_1 \# u_2 \# u_3 \# \dots \# u_{n!} \$, u_2 \# u_3 \# \dots \# u_{n!+1} \#)$$

where $u_i \rightarrow_M u_{i+1}$ for all $i \in [2, n!]$ and $w_1 \rightarrow_M u_2$.

Then $(v_1, a^{n!}, v_2) \in \mathcal{L}(\Psi_1)$ and $(v_1, a^{2 \cdot (n!)}, v_2) \in \mathcal{L}(\Psi_2)$. Suppose that $(v_1, a^{n!}, v_2) \in \mathcal{L}(A_i) \times \mathcal{L}(B_i)$ and that $\delta_{A_i}(q_{i0}, v_1) = p_i$ and $\delta_{A_i}(p_i, a^{n!}) = r_i$ (where δ_{A_i} denotes the transition function of A_i and q_{i0} is the initial state of A_i). Then by the pigeon hole principle on the number of states, we must encounter a loop and hence repeat at least one state while reading $a^{n!}$ from p_i in A_i . Let s_i be the period (length of the loop) $1 \leq s_i \leq n$, such that $\delta_{A_i}(p_i, a^{n!+j_i s_i}) = r_i$ for any $j_i \in \mathbb{N}$. In particular, if we choose $j_i = n! \div s_i$, we get $\delta_{A_i}(p_i, a^{2n!}) = r_i$. Thus, $(v_1, a^{2n!}, v_2)$ is also in $\bigcup_{i=1}^{\ell} L_i \times L'_i$, and which is a contradiction.

Case 2. Suppose M halts in $n > 1$ steps. Then $(w_1 = u_1) \rightarrow_M u_2 \rightarrow_M \dots \rightarrow_M u_n \not\rightarrow_M u_{n+1}$. We construct the REG separator for $\mathcal{L}(\Psi_1)$ and $\mathcal{L}(\Psi_2)$ as follows. Let **config** represent any configuration of M , and let $R_1 = (\text{config} \#)^* \text{config} \$ a^*$, $R_2 = (\text{config} \#)^* \text{config}$. Then R_1, R_2 are regular languages.

We first consider the case when we have $(u, v) \in R_1 \times R_2$ s.t. number of configurations in both u, v is less than n . We construct L_i which checks whether the number of configurations in u is i , and the number of a 's in u is also equal to i . Likewise, we construct language L'_i which checks if the number of configurations in v is i . Then $\bigcup_{i=1}^n L_i \times L'_i$ does not intersect with $\mathcal{L}(\Psi_2)$ and contains $\mathcal{L}(\Psi_1)$. Thus for each $i \in [1, n]$,

$$L_i = \{w_1 \# w_3 \# w_5 \# \dots \# w_{2i-1} \$ a^i \mid w_1, w_3, w_5, \dots, w_{2i-1} \in \Sigma^* \times (Q \times \Sigma) \times \Sigma^*\}$$

$$L'_i = \{w_2 \# w_4 \# w_6 \# \dots \# w_{2i} \mid w_2, w_4, w_6, \dots, w_{2i} \in \Sigma^* \times (Q \times \Sigma) \times \Sigma^*\}$$

Now consider the case when the number of configurations is more than n in u or v , for $(u, v) \in R_1 \times R_2$. It suffices to choose non-deterministically an index $j \leq n$ and check if w_{2j} and w_{2j+1} are not equal and all the configurations before w_{2j+1} and w_{2j} follow the run of M . It is sufficient to check up to n configurations since the run in M has only n configurations in the sequence. For $1 \leq j \leq n$, consider the subset \mathcal{R}_j of $R_1 \times R_2$ defined as $\mathcal{R}_j = \{u_1 \# u_2 \# \dots \# u_{j-1} \# w_{2j+1} (\# \text{config})^* \$ a^*\} \times \{u_2 \# u_3 \# \dots \# u_{j-1} \# u_j (\# \text{config})^* \# \}$ such that $u_j \neq w_{2j+1}$. \mathcal{R}_j is regular since the length of configurations u_j and w_{2j+1} are bounded. $\bigcup_{j=1}^n \mathcal{R}_j \cap \mathcal{L}(\Psi_2) = \emptyset$.

Now consider a pair $(u, v) \in R_1 \times R_2$, where there is no violation as described above in the equality between u_j and w_{2j+1} until $j = n$. Then we cannot continue finding a successor for u_n , as a result of which, such pairs $(u, v) \notin \mathcal{L}(\Psi_1)$, and hence cannot be considered in separator.

This way we remove all words in $\mathcal{L}(\Psi_2)$ and accept all the words in $\mathcal{L}(\Psi_1)$. Hence $(\bigcup_{i=1}^n L_i \times L'_i) \cup (\bigcup_{j=1}^n \mathcal{R}_j)$ is a REG separator. \blacktriangleleft

C Proofs from Section 4

C.1 Proof of Lemma 3

We show that the class of ordered multi pushdown languages (OMPL) form a full trio.

Proof. Let L be an OMPL over alphabet Σ and let \mathcal{A} be the OMPA accepting L . We now show the effective closure wrt the three properties.

- For $B \subseteq \Sigma$, construct an OMPA from \mathcal{A} by replacing transitions on input symbols $a \in \Sigma \setminus B$ with ϵ . The resulting OMPA accepts the B -projection of L . For $B \not\subseteq \Sigma$, the OMPA accepting the emptyset suffices.
- For the B -upward closure of L , add extra transitions to \mathcal{A} as loops on each state on any input symbol from B , with no push/pop operations. The resultant OMPA will accept the B -upward closure of L .
- The third property is the closure wrt intersection of regular languages. It is well-known that OMPL are effectively closed wrt intersection with regular languages.

This shows that OMPLs form a full trio. \blacktriangleleft

C.2 Proof of Lemma 5

In the following, we present the proof to show that each OMPL L has a semilinear Parikh image.

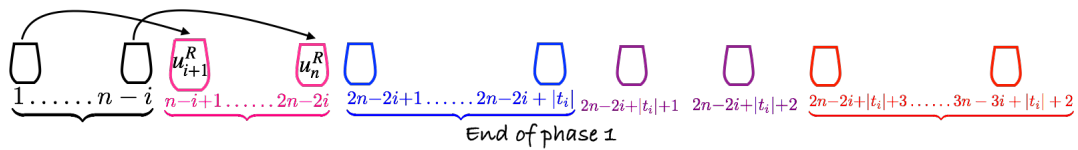
Proof. To see this, first note that using the result of [7], one can construct a Dn -grammar equivalent to the OMPL. (We refer the reader to [7] for the definition of Dn -grammar.) Next, using the result from [11], we know that for each Dn -grammar G there is an underlying context-free grammar G' s.t. for each $w \in L(G)$, there is a $w' \in L(G')$ (and conversely) such that w and w' have the same Parikh image. Thus, the language generated by the Dn -grammar G and the language generated by the context-free grammar G' have the same Parikh image. Since the Parikh image of $L(G')$ is semilinear [40] we obtain that the language of each Dn -grammar is semilinear, giving us the result (since Dn -grammar are equivalent to the OMPL). \blacktriangleleft

C.3 Details for Lemma 8

Proof. For intuition of this proof, readers can refer the proof provided in Lemma 8. Formally, set of states of the OMPA \mathcal{A}_i is states of \mathcal{T}_i in addition with the states $\{q_i^{init}, q_i^2, q_i^4, q_i^{final}\}$ to distinguish phase 1, 2, and 4. Initial state is defined as q_i^{init} .

Phase 1: Being in state q_i^{init} , content of first $(n - i)$ stacks are moved to next $(n - i)$ stacks i.e. content of stack j , is moved to stack $n - i + j$ for all $1 \leq j \leq n - i$. To achieve this, for all j , $1 \leq j \leq n - i$, for all $a \in \Sigma$, we have transition

$$(q_i^{init}, \perp((j - 1) \text{ times}), a, \epsilon, \dots, \epsilon) \rightarrow^\epsilon (q_i^{init}, \epsilon((n - i + j - 1) \text{ times}), a, \epsilon, \dots, \epsilon)$$



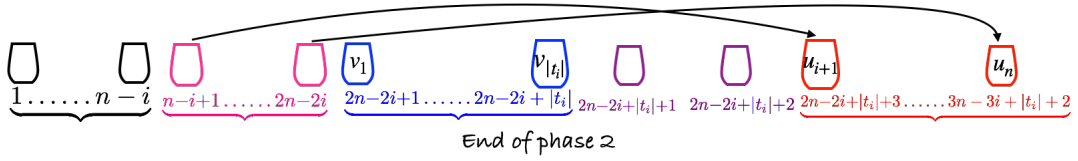
XX:24 On the Separability Problem of String Constraints

Phase 2: After phase 1, stacks indexed 1 to $n-i$ are empty and next $n-i$ stacks contain the valuation of variables x_{i+1} to x_n in reverse. To distinguish this phase, we add a no operation transition from q_i^{init} to q_i^2 . For this, we have a transition

$$(q_i^{init}, \perp((n-i) \text{ times}), \epsilon, \dots, \epsilon) \rightarrow^\epsilon (q_i^2, \epsilon, \dots, \epsilon)$$

In this phase, content of non-empty stacks $(n-i+1), \dots, (2n-2i)$ is moved to last $(n-i)$ stacks indexed $(2n-2i+|t_i|+2)+1, \dots, (2n-2i+|t_i|+2)+(n-i)$. Also, if any of the variable from x_{i+1}, \dots, x_n is present in t_i at position j then the content of the corresponding stack is pushed to the stack $(2n-2i)+j$. It is done with the following transitions. For all k , $1 \leq k \leq (n-i)$, for all $a \in \Sigma$, we have

$$(q_i^2, \perp(n-i+k-1 \text{ times}), a, \epsilon, \dots, \epsilon) \rightarrow^\epsilon (q_i^2, \epsilon((2n-2i) \text{ times}), \\ a \text{ if } t_i[1] = x_{i+k}; \text{ else } \epsilon, \\ a \text{ if } t_i[2] = x_{i+k}; \text{ else } \epsilon, \\ \vdots \\ a \text{ if } t_i[|t_i|] = x_{i+k}; \text{ else } \epsilon, \\ \epsilon((2+k-1) \text{ times}), \\ a, \epsilon, \dots, \epsilon)$$

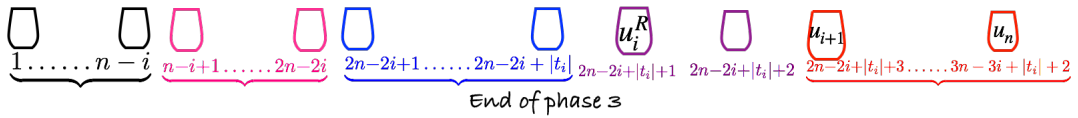


Phase 3: After phase 2, we have valuation of $t_i[j]$ present in stack $(2n-2i+j)$. Also, last $(n-i)$ stacks indexed $(2n-2i+|t_i|+3)$ to $(3n-3i+|t_i|+2)$ contains the valuation of variables x_{i+1}, \dots, x_n respectively. In this phase, we evaluate the valuation of x_i based on valuation of $t_i, \eta(t_i)$. Observe that $\eta(t_i)$ is stored in the sequence in stacks $(2n-2i+1), \dots, (2n-2i+|t_i|)$ and all the stacks before these, are empty. Thanks to OMPA, one can pop the content of these stacks one after another, and find the output of these words produced by \mathcal{T}_i . Output produced is pushed into the stack indexed $(2n-2i+|t_i|+1)$. To begin this process, we use a no operation transition from q_i^2 to p_0 where p_0 belongs to initial states of \mathcal{T}_i :

$$(q_i^2, \perp(2n-2i \text{ times}), \epsilon, \dots, \epsilon) \rightarrow^\epsilon (p_0, \epsilon, \dots, \epsilon)$$

where p_0 is in set of initial state of \mathcal{T}_i . To simulate \mathcal{T}_i on content of these $|t_i|$ stacks, for every transition $(p, (b, a), p')$ of \mathcal{T}_i , and for all j s.t. $1 \leq j \leq |t_i|$ we have following corresponding transitions in \mathcal{A}_i :

$$(p, \perp(2n-2i+(j-1) \text{ times}), a, \epsilon, \dots, \epsilon) \rightarrow^\epsilon (p', \epsilon(2n-2i+|t_i| \text{ times}), b, \epsilon, \dots, \epsilon)$$



Phase 4: After finishing computing value of x_i , if the state of \mathcal{T}_i reached is a final state, \mathcal{A}_i moves to state q_i^4 to start the process of this phase.

$$(p_f, \perp(2n-2i+|t_i| \text{ times}), \epsilon, \dots, \epsilon) \rightarrow^\epsilon (q_i^4, \epsilon, \dots, \epsilon)$$

Proof. The proof is done for the class of PosPTL however it can easily be extended to languages over n -tuples words.

Let I, E be languages over a finite alphabet Σ . Assume I and E are separable by the PosPTL S . Then $I \subseteq S$ and $S \cap E = \emptyset$. Since S is upward closed, $I \uparrow \subseteq S$ and hence $I \uparrow \cap E = \emptyset$.

Assume $I \uparrow \cap E = \emptyset$. We claim that $I \uparrow$ is a PosPTL-separator. We have already that $I \subseteq I \uparrow$ and $I \uparrow \cap E = \emptyset$. Now we will show that $I \uparrow$ is PosPTL: $\min(I \uparrow)$ is a finite set since the subword ordering \preceq is a wqo on Σ^* . Let $\min(I \uparrow) = \{w_1, \dots, w_k\}$. Then $I \uparrow$ can be defined as a finite union of $\{w_i\} \uparrow$ for $i \in \{1, 2, \dots, k\}$. For $w_i = a_{i_1} \dots a_{i_{m_i}}$, with $a_{i_j} \in \Sigma$, $\{w_i\} \uparrow$ is the piece language $\Sigma^* a_{i_1} \Sigma^* \dots \Sigma^* a_{i_{m_i}} \Sigma^*$. Thus, $I \uparrow$ is indeed in PosPTL, being the finite union of piece languages.

To see $I \uparrow \cap E = \emptyset$ iff $I \cap E \downarrow = \emptyset$. Let $w \in I \uparrow \cap E$. Let $w' \preceq w$ be s.t. $w' \in I$. Then $w' \in I \cap E \downarrow$. The converse works similarly. \blacktriangleleft

D.2 PosPTL separability of PDA (Proof of Theorem 14)

We show that PosPTL separability of context free languages is PSPACE-COMplete. Context free languages are equivalently represented by pushdown automata(PDA) or context free grammars(CFG) and the conversion from CFG to PDA and PDA to CFG is in polynomial time. We recall the definition of context free grammar briefly here.

A *Context Free Grammar* is defined by a tuple $G = (N, T, R, S)$ where N and T are finite set of non-terminals and terminals respectively, S is an initial non-terminal. R is a finite set of production rules of the form (i) $A \rightarrow BC$, (ii) $A \rightarrow a$, or (iii) $S \rightarrow \epsilon$ where $A \in N$, $B, C \in N \setminus \{S\}$ and $a \in T$. We define derivation relation \Rightarrow as following: Given $u_1, u_2 \in (N \cup T)^*$, $u_1 \Rightarrow u_2$ iff there exists $A \rightarrow w$ in the production rules set R such that $u_1 = vAv'$ and $u_2 = vww'$ for some $v, v' \in (N \cup T)^*$. Let \Rightarrow^+ denote one or more application of derivation relation \Rightarrow . A string w belongs to the language generated by grammar G iff it can be generated starting from S , applying derivation rules one or more times. Hence, the language generated by the grammar G is $\mathcal{L}(G) = \{w \in T^* \mid S \Rightarrow^+ w\}$.

Proof of Theorem 14 We show that PosPTL separability problem of PDA is PSPACE-COMplete. Given PDA \mathcal{A}_1 and \mathcal{A}_2 , consider their corresponding CFGs G_1, G_2 . $\mathcal{L}(G_1) \uparrow \cap \mathcal{L}(G_2) \neq \emptyset$ iff $\mathcal{L}(G_1) \uparrow \cap \mathcal{L}(G_2) \downarrow \neq \emptyset$. If there is a witness of $\mathcal{L}(G_1) \uparrow \cap \mathcal{L}(G_2)$, the same witness trivially works for $\mathcal{L}(G_1) \uparrow \cap \mathcal{L}(G_2) \downarrow$. Now suppose $w \in \mathcal{L}(G_1) \uparrow \cap \mathcal{L}(G_2) \downarrow$, then there exists $w' \in \mathcal{L}(G_2)$ s.t. $w \preceq w'$. Also, $w' \in \mathcal{L}(G_1) \uparrow$. Hence, we have a witness of the non-emptiness of $\mathcal{L}(G_1) \uparrow \cap \mathcal{L}(G_2)$. We prove that the non-emptiness of $\mathcal{L}(G_1) \uparrow \cap \mathcal{L}(G_2) \downarrow$ is PSPACE – complete.

PSPACE-membership. The non-emptiness check of $\mathcal{L}(G_1) \uparrow \cap \mathcal{L}(G_2) \downarrow$ can be seen to be in PSPACE as follows. We construct PDAs $A_{G_1 \uparrow}, A_{G_2 \downarrow}$ respectively for $\mathcal{L}(G_1) \uparrow$ and $\mathcal{L}(G_2) \downarrow$. This construction takes polynomial time; moreover, $A_{G_1 \uparrow}, A_{G_2 \downarrow}$ use only a bounded stack which is polynomial in the size of the CFG (G_1 or G_2). This fact is rather easy to see for $A_{G_1 \uparrow}$ (Lemma 24), but is quite involved for $A_{G_2 \downarrow}$ (Lemma 25). Assuming the construction of $A_{G_1 \uparrow}, A_{G_2 \downarrow}$, we give an NPSPACE algorithm as follows. Guess a word w , one symbol at a time, and run $A_{G_1 \uparrow}, A_{G_2 \downarrow}$ in parallel. Since both the PDAs require only a polynomially bounded stack size, we need polynomial space to store the information pertaining to the states, stacks and the current input symbol. If $A_{G_1 \uparrow}, A_{G_2 \downarrow}$ accept w , we are done. The PSPACE-membership follows from Savitch's theorem.

PSPACE-hardness. We reduce the halting problem of a Linear Bounded Turing Machine (LBTM) to our problem. Given a LBTM M and a word w , we construct PDAs P_1 and P_2

such that $\mathcal{L}(P_1)\uparrow \cap \mathcal{L}(P_2)\downarrow \neq \emptyset$ iff M accepts w . Our construction is in Appendix D.3.

Bounded Stack Size of $A_{G\uparrow}$, $A_{G\downarrow}$

We first consider the case of $A_{G\uparrow}$. The proof follows by examining the height of derivation trees of words in $\min(\mathcal{L}(G))$, and showing that they have a bounded height. This bound gives the height of the stack size in $A_{G\uparrow}$. The proof of Lemma 24 is in Section D.6.

► **Lemma 24.** *Given a CFG $G = (N, T, R, S)$, the PDA $A_{G\uparrow}$ can be constructed in polynomial time, using a stack whose height is polynomial in the size of G . $\mathcal{L}(A_{G\uparrow}) = \mathcal{L}(G)\uparrow$.*

► **Lemma 25.** *Given CFG $G = (N, T, R, S)$, one can construct PDA $A_{G\downarrow}$ in polynomial time which uses polynomially bounded stack for computation such that $\mathcal{L}(A_{G\downarrow}) = \mathcal{L}(G)\downarrow$.*

Proof. First, we compute a CFG G' using G , and then construct $A_{G\downarrow}$ using G, G' . The construction of G' is described below. We recall the approach described in [16] to compute downward closure, where a regular expression is computed for $\mathcal{L}(G)\downarrow$.

For every language L , let $\alpha(L)$ be the set of terminal symbols occurring in L (hence $\alpha(L) = \emptyset$ iff $L \subseteq \{\epsilon\}$). For $L, L' \subseteq T^*$ we have: (i) $\alpha(L \cup L') = \alpha(LL') = \alpha(L) \cup \alpha(L')$, (ii) $(L \cup L')\downarrow = L\downarrow \cup L'\downarrow$, and (iii) $(LL')\downarrow = L\downarrow L'\downarrow$. For $m \in (N \cup T)^*$, let $L(G, m)$ denote the language generated by G , starting from the word m . For every $A, B \in N$, define:

1. $B <_1 A$ iff $A \Rightarrow_G^+ mBm'$ for some $m, m' \in (N \cup T)^*$,
2. $B <_2 A$ iff $A \Rightarrow_G^+ mBm'Bm''$ for some $m, m', m'' \in (N \cup T)^*$,
3. $B =_1 A$ iff $B <_1 A <_1 B$.

A proof of the first implication in Claim 26 is in Appendix D.4, the second implication has a similar proof.

▷ **Claim 26.** $A <_2 A \Rightarrow L(G, A)\downarrow = (\alpha(L(G, A)))^*$, $A =_1 B \Rightarrow L(G, A)\downarrow = L(G, B)\downarrow$

Computing $L(G, A)\downarrow$. Now, we explain how $L(G, A)\downarrow$ can be computed for any given $A \in N$. If $A <_2 A$, then claim 26 yields the answer. Otherwise, we compute $L(G, A)\downarrow$ in terms of $L(G, B)\downarrow$ for $B <_1 A$ and $B \neq_1 A$, assuming $L(G, B)\downarrow$ is given by previously computed rules for $L(G, B)$. Let $p : A \rightarrow m$ be a production rule. We define the words $R_0(p), R_1(p)$ and $R_2(p)$ as follows, depending on the production rule $p : A \rightarrow m$.

First case: m does not contain any non-terminal $B =_1 A$. We let $R_0(p) := m$, and $R_1(p)$ and $R_2(p)$ be the empty words.

Second case: m contains a unique non-terminal B with $B =_1 A$ and $m = m'Bm''$. We let $R_1(p) := m'$ and $R_2(p) := m''$. Since we assume that $A \not<_2 A$, the word m cannot contain two occurrences of non-terminals $=_1 A$. In this case, $R_0(p)$ is the empty word.

▷ **Claim 27.** For every A such that $A \not<_2 A$, we have:

$$L(G, A)\downarrow = (\cup \alpha(L(G, R_1(p))))^* (\cup L(G, R_0(p))\downarrow) (\cup \alpha(L(G, R_2(p))))^*$$

where the union extend to all production rules p with lefthand side B such that $B =_1 A$.

Since the words $R_0(p), R_1(p)$ and $R_2(p)$ contain only non-terminals C with $C <_1 A$ and $C \neq_1 A$, we have achieved our goal. By this we end the brief recall of the the approach described in [16] to compute downward closure.

Based on the above facts, we can construct intermediate CFG $G' = (N', T', R', S\downarrow)$, which helps to construct PDA for $\mathcal{L}(G)\downarrow$. For each non-terminal $A \in N$, we introduce five non-terminals $A\downarrow, A_l, A_m, A_r$ and A_{alph} as well as dummy terminals a_l, a_r and a_{alph} in G' .

Construction of G' is shown in Algorithm 1 in Appendix D.5. The non-terminal A_{alph} as well as the terminal a_{alph} are used to simulate $\alpha(L(G, A))$. Likewise, the non-terminal A_l (A_r) and the terminal a_l (a_r) are used to simulate $\alpha(L(G, m'))$ ($\alpha(L(G, m''))$) when $m = m' C m''$ in the production $A \rightarrow m$ and $C =_1 A$. The non-terminal A_m is used when $m = A_1 A_2$ for non-terminals $A_1 \neq_1 A, A_2 \neq_1 A$. Finally, A_\downarrow works in place of A , and depending on m in the production $A \rightarrow m$, gets rewritten either as A_{alph} or $A_l A_m A_r$ or $A_1 \downarrow A_2 \downarrow$.

Constructing the PDA $A_{G\downarrow}$ from G, G' . The PDA $A_{G\downarrow} = (Q, T, N', \delta, \{q\}, S\downarrow, \{q\})$ from G' and G is constructed below. This works in the standard way of converting CFGs to PDA, keeping in mind the following. When a_l (resp. a_r, a_{alph}) is the top of the stack, the PDA has a loop over symbols from $\alpha_l(A)$ (resp. $\alpha_r(A), \alpha(L(G, A))$), in whichever state it is, at that time. $\alpha_l(A)$ (resp. $\alpha_r(A)$) is a set of symbols appearing in $\alpha(L(G, m'))$ (resp. $\alpha(L(G, m''))$) for all rules $C \rightarrow m' B m''$ where $C =_1 A =_1 B$. The PDA stays in state q when the top of stack contains a non-terminal A . This non-terminal is popped and the right hand side m of its production rule $A \rightarrow m$ is pushed on the stack. If the top of the stack is i) a terminal from T which also happens to be the next input symbol, then we just pop it ii) a terminal of the form a_{alph}, a_l or a_r , we move to state $q_{A_{\text{alph}}}, q_{A_l}$ or q_{A_r} respectively and loop over $\alpha(L(G, A)), \alpha_l(A)$ or $\alpha_r(A)$ respectively and come back to q non-deterministically. The PDA accepts in q when the stack becomes empty. A_l, A_r and A_{alph} do not produce any further non-terminals while A_m produces non-terminals B where $B <_1 A$ and $B \neq_1 A$. Thus, the stack size is $\leq \mathcal{O}(|N|)$. ◀

D.3 PSPACE-hardness in Theorem 14

Our construction is inspired from [27, Lemma 8.6].

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a linear bounded TM (LBTM). Σ is the input alphabet, Γ is the tape alphabet (contains Σ and B and some extra symbols) and B stands for the blank symbol. M uses at most $p(n)$ space for the input of size n , where $p(n)$ is a polynomial in n . The maximum number of distinct configurations of M can be $\max(|Q|, |\Gamma|)^{p(|w|)+1}$, say k'_{\max} , assuming a configuration is represented as $w_1 q w_2$ where $w_1 w_2$ is a word over Γ that covers the entire tape (thus, we also include the spaces occupied by the blank symbols).

Consider constants $k_{\max} = 2^{\lceil \log k'_{\max} \rceil}$ and $c_{\max} = \log k_{\max}$ to give precise bounds. Without loss of generality we assume that all halting computations of M on w take an even number of steps.

Given the LBTM M , we give the construction of PDAs P_1 and P_2 as follows. The input alphabet of P_1, P_2 is $\Gamma \cup Q \cup \{\#, \$\}$, where $\#, \$$ are new symbols. The stack alphabet for P_1, P_2 contain these symbols as well as symbols $0, 1, \#_1$. $0, 1$ are used to encode binary numbers to keep track of the number of configurations seen so far on any input. The idea is to restrict this number upto k_{\max} .

P_1 and P_2 both generate words of the form $c_1 \# c_2^R \# c_3 \# c_4^R \# \dots \# c_k^R \# \$^{c_{\max}} \dots \# \$^{c_{\max}} \#$ where number of $\#$ s = k_{\max} , satisfying the following conditions.

- (1) each c_i is a valid configuration of M and of size exactly c_{\max} ,
- (2) In P_2 , c_1 is an initial configuration of M on w i.e. $q_0 w B^{c_{\max} - (|w|+1)}$,
- (3) In P_1 , the last configuration before $\$$ is an accepting configuration,
- (4) c_{i+1} is a valid successor configuration of c_i in P_1 for all odd i ,
- (5) c_{i+1} is a valid successor configuration of c_i in P_2 for all even i .

This way, correctness of consecutive odd and even configurations are guaranteed by P_2 and P_1 respectively and $\mathcal{L}(P_1) \cap \mathcal{L}(P_2)$ is the set of accepting computations of M on w padded with copies of word $(\$^{c_{\max}} \#)$ at the right end. The length of the words accepted by P_1 and

P_2 is k_{max} (number of #s) + $c_{max} \cdot k_{max}$ (the size of each c_i or c_i^R is c_{max} , the padding with \$ also has length c_{max} . There are k_{max} of these c_i or padded \$.)

Construction of P_1 .

- (a) Initially we push $\log k_{max}$ zeros (where k_{max} is the maximum possible number of configurations) and a unique symbol $\#_1$ in the stack to keep track of number of #s read so far. $\#_1$ is used as a separator in the stack between the counter values (encoded using 0,1) and the encoded configurations over Γ, Q . Thus, we initialize the value of the binary counter to $\log k_{max}$ zeroes, with the separator $\#_1$ on top.
- (b) Let L be $\{c_1 \# c_2^R \# \mid c_2 \text{ is the successor configuration of } c_1 \text{ in } M\}$. To realize L , we do the following. We read c_1 and check whether it is of the form $w_1 q w'_1$ where $w_1, w'_1 \in \Gamma^*$ and $q \in Q$. While reading c_1 upto q , we push w_1 on its stack. As soon as we find $q \in Q$ in c_1 , we store q in the finite control state and read the next input symbol, say X .
 - If X is $\#$, then there is no successor configuration, and we reject the word denoting that we have reached the end of the tape.
 - If $\delta(q, X) = (p, Y, R)$, then we push Yp onto the stack.
 - If $\delta(q, X) = (p, Y, L)$, let Z be on top of stack, then we replace Z by pZY .

Again while reading w'_1 , we push the symbols read onto the stack. Now the stack content will be (from bottom to top) the encoding of the counter, $\#_1$, and c_2 . After reading the $\#$ after c_1 , we compare each input symbol with the top stack symbol (now we are reading c_2^R in the input). If they differ, there is no next move. If they are equal, we pop the top stack symbol. When the top of stack is $\#_1$, we read $\#$ after c_2^R and accept. While reading c_i (or c_i^R) we must also verify that the length of c_i is exactly c_{max} . We can do this by adding polynomial size counter in the finite control state.

- (c) We also need to increment the counter by 2 for the number of $\#$ (configurations) seen so far. We have read two configurations now in the above step. This can be done by popping the stack content starting from $\#_1$ until we find 0. During this popping, we store in the finite control, the information of how many 1s we have popped. As soon as we find 0 on the top of the stack, replace it with 1 and push back as many number of 1s as we popped previously (this info is stored in the finite control), followed by the separator $\#_1$. We need to do this step again to increment the counter since we saw two configurations (the two #s represent this) in L .

To construct the PDA P_1 , we follow steps of [(a)] i.e., initializing the counter. Next it iterates over steps of [(b)] followed by [(c)], thus reading two consecutive $c_i \# c_{i+1}^R \#$ and incrementing the counter by 2 until we find the configuration of the form $\Gamma^* F \Gamma^*$ or the counter becomes full. If we stop the iteration due to accepting configuration and counter is not full yet (counter is full when we have all 1s in the stack), we pad $\$^{c_{max}} \#$ until it becomes full. Otherwise if we stop the iteration due to the counter reaching k_{max} value before getting accepting configuration, we stop and reject this sequence of configurations.

Construction of P_2 . Similarly, we can construct PDA P_2 : The difference here is that it starts with initial configuration i.e. $q_0 w B^{c_{max}-|w|}$ followed by $\#$ (here also we increment the counter by 1), then it will iterate over $\{c_1^R \# c_2 \# \mid c_2 \text{ is the successor configuration of } c_1 \text{ in } M\}$ and increment the counter by 2 after each iteration. Then it will read accepting configuration non-deterministically (of course, followed by $\#$ and increment in the binary counter) and pad the string with $\$^{c_{max}} \#$ until counter becomes full.

By padding the words accepted with extra $\$^{c_{max}} \#$ s we are enforcing that P_1 and P_2 accepts words of same length. Having the blank symbols B as part of the LBTM configuration is also motivated by this. Indeed, $w \in \mathcal{L}(P_1) \cap \mathcal{L}(P_2)$ iff the LBTM M accepts w . Since the words accepted by P_1, P_2 have same constant length, $\mathcal{L}(P_1) \cap \mathcal{L}(P_2) \neq \emptyset$ iff $\mathcal{L}(P_1) \uparrow \cap \mathcal{L}(P_2) \downarrow \neq \emptyset$.

■ **Algorithm 1** Construction of G'

```

1  foreach non-terminal  $A \in N$  do
2     $\alpha_l(A) := \emptyset, \alpha_r(A) := \emptyset;$ 
3  end
4  foreach non-terminal  $A \in N$  do
5    if  $A <_2 A$  then
6       $A \downarrow \rightarrow A_{\text{alph}}$ 
7    else
8      foreach  $B \in N$  s.t.  $B =_1 A$  do
9        foreach  $p : B \rightarrow m$  do
10         if  $m = A_1 A_2$  for  $A_1 \neq_1 A$  and  $A_2 \neq_1 A$  then
11            $A_m \rightarrow A_1 \downarrow A_2 \downarrow$ 
12         end
13         if  $m = a$  for  $a \in T$  then
14            $A_m \rightarrow a \mid \epsilon$ 
15         end
16         if  $m = m' C m''$  for unique  $C =_1 A$  then
17            $\alpha_l(A) := \alpha(L(G_2, m')) \cup \alpha_l(A), \quad \alpha_r(A) := \alpha(L(G_2, m'')) \cup \alpha_r(A)$ 
18         end
19       end
20     end
21      $A \downarrow \rightarrow A_l A_m A_r, \quad A_l \rightarrow a_l, \quad A_r \rightarrow a_r, \quad A_{\text{alph}} \rightarrow a_{\text{alph}}.$ 
22 end
23 end

```

D.4 Proof of Claim 26

Suppose $w \in L(G, A) \downarrow$, then w is labelled by symbols from $\alpha(L(G, A))$. Clearly $w \in (\alpha(L(G, A)))^*$.

Conversely, assume $w \in (\alpha(L(G, A)))^*$. We prove by induction on the length of w that $w \in L(G, A) \downarrow$. If the length of w is 1, then trivially one can generate any word in $L(G, A)$, which contains w as a subword. Assume the length of w to be $k + 1$. By induction hypothesis, there exists a word $w' \in L(G, A)$ such that $w[:k] \preceq w'$ and w' is generated from A . We construct $w'' \in L(G, A)$ such that $w \preceq w''$. The derivation of w'' from A proceeds as follows. Start with the rule $A \Rightarrow^+ m A m' A m''$, and substitute the first A with w' . For the second A , substitute any word w''' s.t. $A \Rightarrow_G^+ w'''$ s.t. the $k + 1$ st symbol of w , $w[k + 1]$ is contained in w''' . Then we obtain $w \preceq w''$, and hence $w \in L(G, A) \downarrow$.

D.5 Construction of G' and $A_{G \downarrow}$

The PDA $A_{G \downarrow}$ from G, G' . Now we construct the PDA $A_{G \downarrow} = (Q, T, N', \delta, \{q\}, S \downarrow, \{q\})$ from G' and G . This works in the standard way of converting CFGs to PDA, keeping in mind the following. When a_l (resp. a_r, a_{alph}) is the top of the stack, the PDA has a loop over symbols from $\alpha_l(A)$ (resp. $\alpha_r(A), \alpha(L(G, A))$), in whichever state it is, at that time. The PDA stays in state q when the top of stack contains a non-terminal A . This non-terminal is popped the right hand side m of its production rule $A \rightarrow m$ is pushed on the stack. If the top of the stack is i) a terminal from T which also happens to be the next input symbol, then we just pop it ii) a terminal of the form a_{alph}, a_l or a_r , we move to state $q_{A_{\text{alph}}}, q_{A_l}$ or

q_{A_r} respectively and loop over $\alpha(L(G, A))$, $\alpha_l(A)$ or $\alpha_r(A)$ respectively and come back to q non-deterministically. The PDA accepts in q when the stack becomes empty.

Complexity. We analyze the complexity of constructing G' as well as $A_{G\downarrow}$.

- Construction of G' :
 - for every $A, B \in N$, checking $A <_2 B$ reduces to checking membership of $a_1 a_2$ in $L(G_1, B)$ where G_1 is obtained from G by replacing productions $C \rightarrow a$ of G with $C \rightarrow \epsilon$, $C \rightarrow a_1 \mid a_2$. G_1 is constructed in polynomial time from G . Similarly, we can check if $A <_1 B$ and $a \in \alpha(L(G, A))$.
 - for every $A \in N$, five non-terminals and three terminals are introduced. Hence, G' has $5|N|$ non-terminals and $|T| + 3|N|$ terminals, which is again polynomial in the input G' .
 - each step of the algorithm 1 can be computed in at most polynomial time and the algorithm has loops whose sizes range over terminals, non-terminals and production rules of G .
- PDA $A_{G\downarrow}$: Each non-terminal A has corresponding non-terminals A_l , A_r , A_{alph} or A_m in G' . Out of these, A_l , A_r and A_{alph} do not produce any further non-terminals while A_m produces non-terminals B where $B <_1 A$ and $B \neq_1 A$. Thus, the stack size is bounded above by $\mathcal{O}(|N|)$.

D.6 Proof of Lemma 24

Let $G = (N, T, R, S)$, we construct CFG G' which accepts subset of $\mathcal{L}(G)$ and includes $\min(\mathcal{L}(G))$. Clearly $\mathcal{L}(G)\uparrow = \mathcal{L}(G')\uparrow$. Let G' be $(N \times \{0, \dots, |N|\}, T, R', (S, |N|))$ where production rules are defined as following:

- $(A, 0) \rightarrow a$ if $A \rightarrow a$
- $(A, i) \rightarrow (B, i-1)(C, i-1)$ for all $i \in [1, |N|]$ if $A \rightarrow BC$
- $(A, i) \rightarrow (A, i-1)$ for all $i \in [1, |N|]$

We can construct a PDA P for $\mathcal{L}(G')$ in polynomial time. It is easy to see that the PDA uses at most $|N| + 1$ height of the stack. It accepts all words of $\mathcal{L}(G)$ having derivation with at most $|N| + 1$ steps. We claim that $\mathcal{L}(G')$ contains $\min(\mathcal{L}(G))$. If not, then there exists $w \in \min(\mathcal{L}(G))$ with at least $|N| + 2$ derivation steps i.e. some non-terminal A appears at least two times in derivation tree say at level i and $j > i$. If we replace the subtree rooted at level j by subtree at level i , we get a proper subword of w (wlog we assume G is in CNF and it does not have useless symbols and ϵ -productions), which is also in $\mathcal{L}(G)$, contradicting our assumption that $w \in \min(\mathcal{L}(G))$. Finally we can obtain a PDA for $\mathcal{L}(G)\uparrow$ by adding self loops on each state of P for each symbol $a \in \Sigma$ as $\mathcal{L}(G)\uparrow = \mathcal{L}(G')\uparrow$.

D.7 Proof of Lemma 15

Given two SL formulas Ψ and Ψ' , with x_1, \dots, x_n as their set of variables. Let \mathcal{A} and \mathcal{A}' be two OMPAs such that $\mathcal{L}(\mathcal{A}) = \text{Encode}(\mathcal{L}(\Psi))$ and $\mathcal{L}(\mathcal{A}') = \text{Encode}(\mathcal{L}(\Psi'))$. Then, Ψ and Ψ' are separable by an n -PosPTL iff \mathcal{A} and \mathcal{A}' are separable by a PosPTL.

Proof. Let $\mathcal{L}(\mathcal{A})$ and $\mathcal{L}(\mathcal{A}')$ be separable by a PosPTL L over $\Sigma \cup \{\#\}$. Consider a regular language $R \subseteq (\Sigma \cup \{\#\})^*$ containing all words having exactly $(n-1)$ $\#$ s. We first show that $L \cap R$ can be written as a finite Boolean combination, except negation, of languages of the form $L_1 \# L_2 \# \dots \# L_n$ where each L_i is a piece language over Σ .

We prove inductively the representation of $L \cap R$ in terms of positive Boolean combinations of n piece languages over Σ having $\#$ in between them. As a base case, let L be a piece

XX:32 On the Separability Problem of String Constraints

language $\Sigma^* a_1 \Sigma^* \dots \Sigma^* a_k \Sigma^*$. Let S be a finite set containing only the words of the form w such that $a_1 a_2 \dots a_k \preceq w$ and the symbol $\#$ appears exactly $(n-1)$ -times in w . Then $L \cap R$ can be written as $\bigcup_{w \in S} \Sigma^* b_1 \Sigma^* b_2 \dots b_\ell \Sigma^*$, where $w = b_1 \dots b_\ell$, which is of the form

$$\bigcup_{i=1}^{|S|} L_{i1} \# L_{i2} \# \dots \# L_{in} \text{ where } L_{ij} \text{ is a piece language over } \Sigma.$$

Now assume that L is of the form $L_1 \cap L_2$ (resp. $L_1 \cup L_2$). It is easy to see that $L \cap R$ is equivalent to $(L_1 \cap R) \cap (L_2 \cap R)$ (resp. $(L_1 \cap R) \cup (L_2 \cap R)$). Thus we can use our induction hypothesis to show that $L \cap R$ is indeed a Boolean combination (using union and intersection) of languages of the form $L_1 \# L_2 \# \dots \# L_n$ where L_i is a piece language over Σ .

Now, we are ready to construct an n -PosPTL that separates $\mathcal{L}(\Psi)$ and $\mathcal{L}(\Psi')$. First of all, observe that $\mathcal{L}(\mathcal{A}) \subseteq L \cap R$ and $\mathcal{L}(\mathcal{A}') \cap (L \cap R) = \emptyset$ (from the definition of \mathcal{A} and \mathcal{A}'). Since $L \cap R$ can be written as a positive Boolean combination of languages of the form $L_1 \# L_2 \# \dots \# L_n$ where L_i 's are piece languages over Σ , we can easily construct an n -PosPTL over Σ that separates the two formulas Ψ and Ψ' as a finite positive Boolean combination of the product of these L_i s. This is similar to Lemma 10, where we used the fact that a finite Boolean combination of the product of piece languages is a PTL over an n -tuple alphabet. The proof of the other direction follows similarly. ◀

D.8 Proof of claim in section 5.2, line 421

Converting SL with functional transducers to right-sided SL

► **Lemma 28.** *Let Ψ be an SL formula where all relational constraints have only functional transducers. Then we can obtain a right-sided SL formula Ψ' such that Ψ' is satisfiable iff Ψ is.*

Proof. Assume that the SL formula Ψ with functional transducers is given as

$$\Psi = \bigwedge_{i=1}^n x_i \in \mathcal{L}(\mathcal{A}_i) \wedge \bigwedge_{i=1}^k \varphi_i$$

such that φ_i is of the form $\mathcal{T}_i(t_i) = x_i$, and \mathcal{T}_i defines a (partial) function. Without loss of generality, assume that each t_i is a sequence of variables. By the SL condition, if a variable x_j appears in t_i then $j > i$. Notice that Ψ need not be rightsided, since we can have φ_h as the constraint $x_h = \mathcal{T}_h(t_h)$ with $t_h = x_j$, $j > h$ and also φ_j being $x_j = \mathcal{T}_j(t_j)$ violating the condition that none of the output variables x_1, \dots, x_k can be present in any of the inputs t_1, \dots, t_k . The variables x_{k+1}, \dots, x_n are independent and do not appear in the left side (or outputs) of any of the transductions.

We first explain the idea of the proof. We construct a formula Ψ' in which the variables appearing in the inputs terms t'_1, \dots, t'_k of all the transductions in Ψ' are independent variables of Ψ' . Thus, each of t'_1, \dots, t'_k will be sequences of variables from x_{k+1}, \dots, x_n . The idea is to apply substitution inductively starting from φ_k to φ_1 , obtaining in Ψ' , the constraints $\varphi'_k, \dots, \varphi'_1$ s.t. Ψ' are rightsided. Note that by the SL definition, t_k can have only variables from x_{k+1}, \dots, x_n . Now consider φ_{k-1} given by $x_{k-1} = \mathcal{T}_{k-1}(t_{k-1})$. We know that t_{k-1} can have only variables from x_k, x_{k+1}, \dots, x_n . We can replace all occurrences of x_k in t_{k-1} with $\mathcal{T}_k(t_k)$ obtaining $x_{k-1} = \mathcal{T}_{k-1}(t_{k-1}[x_k \mapsto \mathcal{T}_k(t_k)])$. Likewise, we can replace occurrences of x_{k-1}, x_k in t_{k-2} respectively with $\mathcal{T}_{k-1}(t_{k-1})$ and $\mathcal{T}_k(t_k)$. We proceed iteratively for φ_{k-3} to φ_1 .

We give the formal construction by induction on n . For the base case, φ'_k is same as φ_k since t_k is over the independent variables $\{x_{k+1}, \dots, x_n\}$. The same holds for all φ'_h s.t. the input t_h is over $\{x_{k+1}, \dots, x_n\}$.

For the inductive step, let j be the largest index s.t. the input t_j in φ_j contains dependent variables (from $\{x_{j+1}, \dots, x_k\}$). Let φ_j be given by $x_j = \mathcal{T}_j(t_j)$ where $t_j = t_j[1] \dots t_j[m]$, where $t_j[1], \dots, t_j[m] \in \{x_{j+1}, \dots, x_k, x_{k+1}, \dots, x_n\}$. Consider an accepting run in \mathcal{T}_j visiting states q_0, \dots, q_m s.t. the part of the run between q_{i-1} and q_i processes the input $t_j[i]$. If there are multiple accepting runs for the same input word in \mathcal{T}_j , the sequence s of states q_0, \dots, q_m can differ, but the output will be the same since \mathcal{T}_j is functional. Let $\mathcal{T}_j(q_0, q_1), \dots, \mathcal{T}_j(q_{m-1}, q_m)$ represent transducers induced from \mathcal{T}_j , s.t. $\mathcal{T}_j(q_0, q_1)$ reads the first variable $t_j[1]$ and produces output (say o_1), $\mathcal{T}_j(q_1, q_2)$ reads the second variable $t_j[2]$ and produces output (say o_2), and so on till $\mathcal{T}_j(q_{m-1}, q_m)$ reads the m th variable $t_j[m]$ and produces output (say o_m). Let $\mathcal{T}_j^{s,i}$ denote the induced transducer $\mathcal{T}_j(q_{i-1}, q_i)$ for $1 \leq i \leq m$ (i.e., the transducer \mathcal{T}_j with q_{i-1} as initial state and q_i as a final one). Then, for a fixed sequence of states q_0, \dots, q_m , we can write x_j as $\mathcal{T}_j^{s,1}(t_j[1])\mathcal{T}_j^{s,2}(t_j[2]) \dots \mathcal{T}_j^{s,m}(t_j[m])$. Note that the concatenation of transducers $\mathcal{T}_j^{s,1}, \dots, \mathcal{T}_j^{s,m}$ over inputs $t_j[1], \dots, t_j[m]$ can be replaced by a single transducer \mathcal{T}_j' over the input $t_j = t_j[1] \dots t_j[m]$ producing output $o_1 \dots o_m$, since transducers are closed under concatenation.

Assume that $t_j[\ell] = x_p$ is a dependent variable, $p \in \{j+1, \dots, k\}$. Then we can replace $\mathcal{T}_j^{s,\ell}(t[\ell])$ with $\mathcal{T}_j^{s,\ell}[\mathcal{T}_p(t_p)]$. Now, we apply such substitution to all the variables appearing in t . This will result in that t' is a sequence over independent variables $\{x_{k+1}, \dots, x_n\}$.

Thus, we can rewrite x_j as $\mathcal{T}_j^{s,1}(\mathcal{T}'_{p_1}(t_{p_1}))\mathcal{T}_j^{s,2}(\mathcal{T}'_{p_2}(t_{p_2})) \dots \mathcal{T}_j^{s,m}(\mathcal{T}'_{p_m}(t_{p_m}))$ where p_1, \dots, p_m is defined such that $t_j[\ell] = x_{p_\ell}$ for all $\ell \in \{1, \dots, m\}$, and \mathcal{T}'_{p_i} is the transducer given by \mathcal{T}_{p_i} in case $p_i \in \{j+1, \dots, k\}$, and is the identity otherwise. Functional transducers are closed under composition, hence we can obtain a functional transducer \mathcal{T}_j'' equivalent to each of the compositions $\mathcal{T}_j^{s,i} \circ \mathcal{T}'_{p_i}$, for $1 \leq i \leq m$. This expression for x_j is for a fixed accepting path through states q_0, \dots, q_m . Considering the general case of any accepting path over an accepting sequence $q_{i_0} \dots q_{i_m}$, we can write φ_j'' as

$$\varphi_j'' = \bigvee_{s=q_{i_0}, \dots, q_{i_m}} [x_j = \mathcal{T}_j^{s,1}(\mathcal{T}'_{p_1}(t_{p_1}))\mathcal{T}_j^{s,2}(\mathcal{T}'_{p_2}(t_{p_2})) \dots \mathcal{T}_j^{s,m}(\mathcal{T}'_{p_m}(t_{p_m}))]$$

Notice again that $t_{p_1}t_{p_2} \dots t_{p_m}$ is a sequence over independent variables, and the union and concatenation above can be replaced by a single transducer \mathcal{T}_j^s . \mathcal{T}_j^s is the transducer obtained by the concatenation of the following transducers $(\mathcal{T}_j^{s,1}(\mathcal{T}'_{p_1})) \cdot \{(\epsilon, \$)\} \cdot (\mathcal{T}_j^{s,2}(\mathcal{T}'_{p_2})) \cdot \{(\epsilon, \$)\} \cdot \dots \cdot \{(\epsilon, \$)\} \cdot (\mathcal{T}_j^{s,m}(\mathcal{T}'_{p_m}))$ where $\$$ is a special symbol not in Σ . Now let \mathcal{T}_j' be the transducer obtained by taking the union of \mathcal{T}_j^s where s is a sequence of states of \mathcal{T}_j of length m . Now we can rewrite φ_j'' as φ_j''' defined by

$$x_j = \mathcal{T}_j'(t_{p_1} \$ t_{p_2} \$ \dots \$ t_{p_m})$$

We can now replace the occurrence of $\$$ by a fresh variable z not appearing in Ψ to obtain the constraint φ_j' defined by

$$x_j = \mathcal{T}_j'(t_{p_1} z t_{p_2} z \dots z t_{p_m})$$

We now proceed with the next largest j s.t. the input of φ_j is replaced by φ_j' and has now a dependent variable. We repeat this procedure until we get rid of all the dependant variables as input of the relational constraints. Let Ψ'' the resulting formula.

$$\Psi'' = \bigwedge_{i=1}^n x_i \in \mathcal{L}(\mathcal{A}_i) \wedge \bigwedge_{i=1}^k \varphi_i'$$

XX:34 On the Separability Problem of String Constraints

We can now define the formula Ψ' as the conjunction of Ψ'' and $z \in \{\$\}$.

Based on the construction of Ψ' , we have the following claims.

► **Lemma 29.** *Ψ' is a rightsided SL formula.*

Proof. By construction, we have

$$\Psi' = z \in \{\$\} \wedge \bigwedge_{i=1}^n x_i \in \mathcal{L}(\mathcal{A}_i) \wedge \bigwedge_{i=1}^k \varphi'_i$$

The variables z, x_{k+1}, \dots, x_n are clearly independent in Ψ' as well. Consider a φ'_j . As described above, we know that the input of the transduction φ'_j is a sequence over independent variables (thanks to composition of transducers). All input variables are among $\{z, x_{k+1}, \dots, x_n\}$, the independent variables. Hence Ψ' is rightsided. ◀

► **Lemma 30.** *The formula Ψ' constructed is satisfiable iff Ψ is.*

Proof. The equivalence of each φ'_j with φ_j ensures that we have the same functions computed by the transductions, and hence all the inputs and outputs agree in each of the old and new transductions. This, along with the fact that independent variables remain untouched (with the exception of the fresh variable z which has as evaluation $\$$), ensure that we have the same valuation for x_1, \dots, x_n in both Ψ and Ψ' . ◀

Thus, we have shown that, given a SL string constraint having only functional transducers, we can compute an equisatisfiable rightsided SL string constraint. ◀

D.9 Details for Lemma 19

Given a right-sided SL formula Ψ over Σ , with x_1, x_2, \dots, x_n as its set of variables, we wish to construct, in polynomial time, a 2NFT \mathcal{A}_Ψ such that $\mathcal{R}(\mathcal{A}_\Psi) = \{(u_1 \# u_2 \# \dots \# u_n, w_1 \# w_2 \# \dots \# w_n) \mid u_1 \# u_2 \# \dots \# u_n \in \text{Encode}(\mathcal{L}(\Psi)) \text{ and } w_i = u_i \text{ if } x_i \text{ is an independent variable}\}$.

To recall, Ψ is of the form $\bigwedge_{i=1}^n y_i \in \mathcal{L}(\mathcal{A}_i) \wedge \bigwedge_{i=1}^k (y_i, t_i) \in \mathcal{R}(\mathcal{T}_i)$ with y_1, \dots, y_n is a permutation of x_1, \dots, x_n . $\pi : [1, n] \rightarrow [1, n]$ is the mapping that associates to each index $i \in [1, n]$, the index $j \in [1, n]$ s.t. $x_i = y_j$ (or $x_i = y_{\pi(i)}$).

As already mentioned above, given n blocks separated by $\#$, \mathcal{A}_Ψ treats block i as valuation of x_i if x_i is an independent variable, for all i . For other (dependent) variables, say x_j , \mathcal{A}_Ψ computes its valuation based on the valuation of variables given in input blocks. \mathcal{A}_Ψ considers the ordering of variables as x_1, \dots, x_n for input and output. Thus if the input is $u_1 \# u_2 \# \dots \# u_n$, then \mathcal{A}_Ψ tries to find satisfying assignment η for Ψ , and output for this input is given as $\eta(x_1) \# \dots \# \eta(x_n)$ s.t. $\eta \models \Psi$ and $\eta(x_i) = u_i$ for all independent variables i .

Working of \mathcal{A}_Ψ : It produces valuation of x_1, x_2, \dots, x_n in sequence. Depending on if x_i is dependent or independent variable, process given in Lemma 19 is followed by \mathcal{A}_Ψ . To execute the process, \mathcal{A}_Ψ uses following set of states and transitions.

Set of states:

- For each x_i , if x_i is an independent variable, we have following set of states used to produce x_i in the output:

- $\{(q_-(x_i), r) \mid r \text{ is an initial state of } A_{\pi(i)}\}$
The first set indicates that now we want to output value of x_i in the run, which is present in i^{th} block in the input. So for that we first move to \vdash , and \mathcal{A}_Ψ remains in state $(q_-(x_i), r)$ while performing this operation, i.e. moving toward left end.
- $\{(q_j(x_i), r) \mid 0 \leq j < i, r \text{ is an initial state of } A_{\pi(i)}\}$
These states are visited after $(q_-(x_i), r)$, specifically when \mathcal{A}_Ψ is in $(q_j(x_i), r)$, it represents that we have already scanned j blocks in the input, and currently reading $(j+1)$ th block.
- $\{(q_i(x_i), r) \mid r \in Q_{A_{\pi(i)}}\}$
With the help of above states, \mathcal{A}_Ψ reaches the desired i th block in the input and in the beginning of this block, state is $(q_i(x_i), r)$, where r is initial state of $A_{\pi(i)}$. Later \mathcal{A}_Ψ simulates the transitions of $A_{\pi(i)}$ to check the membership constraint $y_{\pi(i)} \in A_{\pi(i)}$ in the second component of states of \mathcal{A}_Ψ with the help of third set.
- For each x_i , if x_i is a dependent variable, we have following set of states used to produce x_i in the output:
 - $\{(q_-(x_i, x_{i_j}, j)) \mid t_{\pi(i)}[j] = x_{i_j}, 1 \leq j \leq t_{\pi(i)}, p \in Q_{\mathcal{T}_{\pi(i)}}, r \in Q_{A_{\pi(i)}}\}$
The first set of states are similar to above case. \vdash present in state name indicates that \mathcal{A}_Ψ is in process of moving left to reach \vdash and the next, we want to read x_{i_j} from $t_{\pi(i)}$, position j (i.e. $t_{\pi(i)}[j] = x_{i_j}$), to produce part of valuation of x_i . The reading of value of x_{i_j} must start from states p in \mathcal{T}_i , while its output must match with the word starting from r in $A_{\pi(i)}$. Here the third parameter j is important since it remembers the progress of $t_{\pi(i)}$, how many variables from it are already processed.
 - $\{(q_k(x_i, x_{i_j}, j), (p, r)) \mid 0 \leq k \leq i_j, t_{\pi(i)}[j] = x_{i_j}, p \in Q_{\mathcal{T}_{\pi(i)}}, r \in Q_{A_{\pi(i)}}\}$
The second set of states are visited to count the number of $\#$ s before reading value of x_{i_j} . Once this count (presented by a subscript of q) reaches $i_j - 1$, we are in block i_j . Then \mathcal{A}_Ψ simulates the transitions of $\mathcal{T}_{\pi(i)}$ starting from p and in parallel simulates the transitions of $A_{\pi(i)}$ starting from state r on the output produced by $\mathcal{T}_{\pi(i)}$.
- special state q^{final} to mark the end.

Initial states:

$$\{(q_0(x_1, x_{1_1}, 1), (p_0, r_0)) \mid x_1 \text{ is dependent, } t_{\pi(1)} = x_{1_1}, p_0, r_0 \text{ are initial states of } \mathcal{T}_{\pi(1)}, A_{\pi(1)}\}$$

$$\cup \{(q_0(x_1), r_0) \mid x_1 \text{ is independent, } r_0 \text{ initial of } A_{\pi(1)}\}$$

Final states: $\{q^{final}\}$

Set of transitions: For all $i \in [1, n]$:

- If x_i is independent variable, and r is initial state of $A_{\pi(i)}$:
 1. $(q_-(x_i), r) \xrightarrow[\alpha|\epsilon, -1]{\#|\epsilon, -1} (q_-(x_i), r)$
This loop helps to reach \vdash in the input.
 2. Once we reach the left end \vdash from $(q_-(x_i), r)$:
 $(q_-(x_i), r) \xrightarrow{\vdash|\epsilon, +1} (q_0(x_i), r)$
 3. To read x_i from input, we need to skip i many $\#$ s using following transitions. For all $1 \leq j \leq i - 2$:
 $(q_j(x_i), r) \xrightarrow{\alpha|\epsilon, +1} (q_j(x_i), r)$ for any $\alpha \in \Sigma$
 $(q_j(x_i), r) \xrightarrow{\#|\epsilon, +1} (q_{j+1}(x_i), r)$

XX:36 On the Separability Problem of String Constraints

$$(q_{i-1}(x_i), r) \xrightarrow{\alpha|\epsilon, 0} (q_i(x_i), r) \text{ for any } \alpha \in \Sigma \cup \{\#, \vdash\}$$

These transitions help in counting the number of blocks separated by $\#$. It stops when \mathcal{A}_Ψ reaches i th block.

- Once \mathcal{A}_Ψ reaches the i th block, it needs to simulate the transitions of $A_{\pi(i)}$: If x_i is independent variable, and current state is r while checking membership in $A_{\pi(i)}$:

$$(q_i(x_i), r) \xrightarrow{\alpha|\alpha, +1} (q_i(x_i), r') \text{ if } (r, \alpha, r') \in \delta_{A_{\pi(i)}}, \text{ for any } \alpha \in \Sigma$$

Above transition simulates $A_{\pi(i)}$ in second component while producing the same output. After it finishes the block x_i , it moves ahead to produce x_{i+1} next producing separator $\#$ in the output:

1. If x_{i+1} is independent variable, and r_0 is initial state of $A_{\pi(i+1)}$, r is final state of $A_{\pi(i)}$:

$$(q_i(x_i), r) \xrightarrow{\#|\#, -1} (q_{\vdash}(x_{i+1}), r_0)$$

2. If x_{i+1} is dependent variable, and r_0 is initial state of $A_{\pi(i+1)}$, p_0 is initial state of $\mathcal{T}_{\pi(i+1)}$, $t_{\pi(i)}[1] = x_{i+1}$, r is final state of $A_{\pi(i)}$:

$$(q_i(x_i), r) \xrightarrow{\#|\#, -1} (q_{\vdash}(x_{i+1}, x_{i+1}, 1), (p_0, r_0))$$

- If x_i is dependent variable, r_0 is initial state of $A_{\pi(i)}$, p_0 is initial state of $\mathcal{T}_{\pi(i)}$, and $t_{\pi(i)}[1] = x_{i_1}$:

$$(q_{\vdash}(x_i, x_{i_1}, 1), (p_0, r_0)) \xrightarrow{\alpha|\epsilon, -1 \atop \#|\epsilon, -1} (q_{\vdash}(x_i, x_{i_1}, 1), (p_0, r_0)) \text{ for any } \alpha \in \Sigma$$

This transition moves \mathcal{A}_Ψ towards \vdash . Once \vdash is reached, it starts counting the number of blocks separated by $\#$ s.

$$(q_{\vdash}(x_i, x_{i_1}, 1), (p_0, r_0)) \xrightarrow{\vdash|\epsilon, +1} (q_0(x_i, x_{i_1}, 1), (p_0, r_0))$$

- If x_i is dependent variable, and $t_{\pi(i)}[j] = x_{i_j}$, for any $0 \leq k \leq i_j - 2$: (Counting of $\#$ separated blocks)

$$(q_k(x_i, x_{i_j}, j), (p, r)) \xrightarrow{\alpha|\epsilon, +1} (q_k(x_i, x_{i_j}, j), (p, r)) \text{ for any } \alpha \in \Sigma$$

$$(q_k(x_i, x_{i_j}, j), (p, r)) \xrightarrow{\#|\epsilon, +1} (q_{k+1}(x_i, x_{i_j}, j), (p, r))$$

- If x_i is dependent variable, and $t_{\pi(i)}[j] = x_{i_j}$, $k = i_j - 1$ (When the desired block is reached in the input):

$$(q_k(x_i, x_{i_j}, j), (p, r)) \xrightarrow{\alpha|\epsilon, 0} (q_{k+1}(x_i, x_{i_j}, j), (p, r))$$

- If x_i is dependent variable, and $t_{\pi(i)}[j] = x_{i_j}$, $k = i_j$ (Simulating transitions of $\mathcal{T}_{\pi(i)}$ on input, while $A_{\pi(i)}$ on output) :

1. if $(p, (b, a), p') \in \delta_{\mathcal{T}_{\pi(i)}}$ and $(q, b, q') \in \delta_{A_{\pi(i)}}$, then

$$(q_k(x_i, x_{i_j}, j), (p, r)) \xrightarrow{a|b, +1} (q_k(x_i, x_{i_j}, j), (p', r'))$$

2. if $(p, (b, \epsilon), p') \in \delta_{\mathcal{T}_{\pi(i)}}$ and $(q, b, q') \in \delta_{A_{\pi(i)}}$, then

$$(q_k(x_i, x_{i_j}, j), (p, r)) \xrightarrow{a|b, 0} (q_k(x_i, x_{i_j}, j), (p', r'))$$

3. if $|t_{\pi(i)}| > j$, $t_{\pi(i)}[j+1] = x_{i_{j+1}}$ (If the current variable is not the last in the term, \mathcal{A}_Ψ proceeds with next variable in term)

$$(q_k(x_i, x_{i_j}, j), (p, r)) \xrightarrow{\#|\epsilon, -1} (q_-(x_i, x_{i_{j+1}}, j+1), (p, r))$$

4. if $|t_{\pi(i)}| = j$, r and p are final states of $A_{\pi(i)}$ and $\mathcal{T}_{\pi(i)}$ respectively (If the current variable is the last in the term, \mathcal{A}_Ψ has finished producing x_i and proceeds with next variable x_{i+1} after producing $\#$ as separator),

- a. if x_{i+1} is an independent variable, and r_0 is initial state of $A_{\pi(i+1)}$

$$(q_k(x_i, x_{i_j}, j), (p, r)) \xrightarrow{\#|\#, -1} (q_-(x_{i+1}), r_0)$$

- b. if x_{i+1} is dependent variable, and r_0 is initial state of $A_{\pi(i+1)}$, p_0 is initial state of $\mathcal{T}_{\pi(i+1)}$, $t_{\pi(i)}[1] = x_{i+1_1}$

$$(q_k(x_i, x_{i_j}, j), (p, r)) \xrightarrow{\#|\#, -1} (q_-(x_{i+1}, x_{i+1_1}, 1), (p_0, r_0))$$

- If x_n is independent variable and r is a final state of $A_{\pi(n)}$

$$(q_n(x_n), r') \xrightarrow[\#|\epsilon, +1]{-|\epsilon, +1} q^{final}$$

- If x_n is dependent variable, and $t_{\pi(n)}[j] = x_{n_j}$, $k = n_j$, $|t_{\pi(n)}| = j$, r and p are final states of $A_{\pi(n)}$ and $\mathcal{T}_{\pi(n)}$ respectively,

$$(q_k(x_n, x_{n_j}, j), (p, r)) \xrightarrow[\neg|\epsilon, +1]{\#|\epsilon, +1} q^{final}$$

- finally, since the acceptance condition is to reach final state after reading \neg :

$$q^{final} \xrightarrow[\neg|\epsilon, +1]{\alpha|\epsilon, +1} q^{final}$$