# A Logic for Document Spanners*

**Dominik D. Freydenberger**

**University of Bayreuth, Bayreuth, Germany**

## Abstract

Document spanners are a formal framework for information extraction that was introduced by Fagin, Kimelfeld, Reiss, and Vansummeren (PODS 2013, JACM 2015). One of the central models in this framework are core spanners, which are based on regular expressions with variables that are then extended with an algebra. As shown by Freydenberger and Holldack (ICDT 2016), there is a connection between core spanners and $\mathsf{EC}^{\mathsf{reg}}$, the existential theory of concatenation with regular constraints. The present paper further develops this connection by defining $\mathsf{SpLog}$, a fragment of $\mathsf{EC}^{\mathsf{reg}}$ that has the same expressive power as core spanners. This equivalence extends beyond equivalence of expressive power, as we show the existence of polynomial time conversions between this fragment and core spanners. This even holds for variants of core spanners that are based on automata instead of regular expressions. Applications of this approach include an alternative way of defining relations for spanners, insights into the relative succinctness of various classes of spanner representations, and a pumping lemma for core spanners.

**1998 ACM Subject Classification** H.2.1 Data models, H.2.4 Textual databases, Relational databases, Rule-based databases, F.4.3 Classes defined by grammars or automata, Decision problems, F.1.1 Relations between models, F.4.m Miscellaneous

**Keywords and phrases** Information extraction, document spanners, word equations, regex, descriptional complexity

**Digital Object Identifier** 10.4230/LIPIcs..2017.

## 1 Introduction

Fagin, Kimelfeld, Reiss, and Vansummeren [12] introduced *document spanners* as a formal framework for information extraction. Document spanners formalize the query language AQL that is used in IBM's SystemT. On an intuitive level, document spanners can be understood as a generalized form of searching in a text $w$: In its basic form, search can be understood as taking a search term $u$ (or a regular expression $\alpha$) and a word $w$, and computing all intervals of positions of $w$ that contain $u$ (or a word from $\mathcal{L}(\alpha)$). These intervals are called *spans*. Spanners generalize searching by computing *relations over spans* of $w$.

In order to define spanners, [12] introduced *regex formulas*, which are regular expressions with variables. Each variable $x$ is connected to a subexpression $\alpha$, and when $\alpha$ matches a subword of $w$, the corresponding span is stored in $x$ (this behaves like the capture groups that are often used in real world implementation of search-and-replace functionality). *Core spanners* combine these regex formulas with the algebraic operators projection, union, join (on spans), and string equality selection.

Freydenberger and Holldack [14] connected core spanners to $\mathsf{EC}^{\mathsf{reg}}$, the existential theory of concatenation with regular constraints. Described very informally, $\mathsf{EC}^{\mathsf{reg}}$ is a logic that combines equations on words (like $x\mathtt{ab}y = y\mathtt{ba}x$) with positive logical connectives, and regular languages that constrain variable replacement. In particular, [14] showed that every

* Supported by Deutsche Forschungsgemeinschaft (DFG) under grant FR 3551/1-1.

© Dominik D. Freydenberger;
licensed under Creative Commons License CC-BY
Full version of a paper that was accepted at ICDT 2017. Last update: July 6, 2016.

core spanner can be converted into an $\mathsf{EC}^{\mathsf{reg}}$-formula, which can then be used to decide satisfiability. Furthermore, while every $\mathsf{EC}^{\mathsf{reg}}$-formula can be converted into an equisatisfiable core spanner, the resulting spanner cannot be used to evaluate the formula (as, due to details of the encoding, the input word $w$ of the spanner needs to encode the formula).

This paper further explores the connection of core spanners and $\mathsf{EC}^{\mathsf{reg}}$. As main conceptual contribution, we introduce SpLog (short for *spanner logic*), a natural fragment of $\mathsf{EC}^{\mathsf{reg}}$ that has the same expressive power as core spanners. In contrast to the PSPACE-complete combined complexity of $\mathsf{EC}^{\mathsf{reg}}$-evaluation, the combined complexity of SpLog-evaluation is NP-complete, and its data complexity is in NL. As main technical result, we demonstrate the existence of polynomial time conversions between SpLog and spanner representations (in both directions), even if the spanners are defined with automata instead of regex formulas.

As a consequence, SpLog can augment (or even replace) the use of regex formulas or automata in the definition of core spanners. Moreover, this shows that the PSPACE upper bounds from [14] for deciding satisfiability and hierarchicality of regex formula based spanners apply to automata based spanners as well. In addition to this, we adapt a pumping lemma for word equations to SpLog (and, hence, to core spanners). The main result also provides insights into the relative succinctness of classes of automata based spanners: While there are exponential trade-offs between various classes of automata, these differences disappear when adding the algebraic operators.

From a more general point of view, this paper can also be seen as an attempt to connect spanners to the research on equations on words and on groups (cf. Diekert [8, 7] for surveys), where $\mathsf{EC}^{\mathsf{reg}}$ has been studied as a natural extension of word equations. We shall see that SpLog is a natural fragment of $\mathsf{EC}^{\mathsf{reg}}$: On an informal level, SpLog has to express relations on a word $w$ without using additional working space (which explains the friendlier complexity of evaluation, in comparison to $\mathsf{EC}^{\mathsf{reg}}$). This gives us reason to expect that SpLog can be applied to other models, like graph databases (as a related example of an application of $\mathsf{EC}^{\mathsf{reg}}$ for graph databases, Barceló and Muñoz [1] use a restricted class of $\mathsf{EC}^{\mathsf{reg}}$-formulas for which data complexity is also in NL).

This paper is structured as follows: Section 2 gives the definitions of spanners and of $\mathsf{EC}^{\mathsf{reg}}$, as well as a few preliminary results. Section 3 introduces SpLog and connects it to spanners. We then examine properties of SpLog: Section 4 discusses how SpLog can be used to express relations, while Section 5 adapts an EC-inexpressibility result to SpLog. Section 6 concludes the paper. Most of the proofs were moved to the Appendix.

## 2 Preliminaries

Let $\Sigma$ be a fixed finite alphabet of *(terminal) symbols*. Except when stated otherwise, we assume $|\Sigma| \geq 2$. Let $\Xi$ be an infinite alphabet of *variables* that is disjoint from $\Sigma$. We use $\varepsilon$ to denote the *empty word*. For every word $w$ and every letter $a$, let $|w|$ denote the length of $w$, and $|w|_a$ the number of occurrences of $a$ in $w$. A word $x$ is a *subword* of a word $y$ if there exist words $u, v$ with $y = uxv$. We denote this by $x \sqsubseteq y$; and we write $x \not\sqsubseteq y$ if $x \sqsubseteq y$ does not hold. For words $x, y, z$ with $x = yz$, we say that $y$ is a *prefix* of $x$, and $z$ is a *suffix* of $x$. A prefix or suffix $y$ of $x$ is *proper* if $x \neq y$. For every $k \geq 0$, a *$k$-ary word relation (over $\Sigma$)* is a subset of $(\Sigma^*)^k$. Given a nondeterministic finite automaton (NFA) $A$ (or a regular expression $\alpha$), we use $\mathcal{L}(A)$ (or $\mathcal{L}(\alpha)$) to denote its language. In NFAs, we allow the use of $\varepsilon$-transitions (this model is also called $\varepsilon$-NFA in literature).

The remainder of this section contains the two models that this paper connects: Document spanners in Section 2.1, and $\mathsf{EC}^{\mathsf{reg}}$ in Section 2.2.

## 2.1 Document Spanners

### 2.1.1 Primitive Spanner Representations

Let $w := a_1 a_2 \cdots a_n$ be a word over $\Sigma$, with $n \geq 0$ and $a_1, \ldots, a_n \in \Sigma$. A *span of* $w$ is an interval $[i, j\rangle$ with $1 \leq i \leq j \leq n+1$ and $i, j \geq 0$. For each span $[i, j\rangle$ of $w$, we define $w_{[i,j\rangle} := a_i \cdots a_{j-1}$. That is, each span describes a subword of $w$ by its bounding indices.

▶ **Example 1.** Let $w := \mathtt{aabbcabaa}$. As $|w| = 9$, both $[3,3\rangle$ and $[5,5\rangle$ are spans of $w$, but $[10,11\rangle$ is not. As $3 \neq 5$, the first two spans are not equal, even though $w_{[3,3\rangle} = w_{[5,5\rangle} = \varepsilon$. The whole word $w$ is described by the span $[1,10\rangle$.

Let $V \subset \Xi$ be finite, and let $w \in \Sigma^*$. A $(V,w)$*-tuple* is a function $\mu$ that maps each variable in $V$ to a span of $w$. If context allows, we write $w$-tuple instead of $(V,w)$-tuple. A set of $(V,w)$-tuples is called a $(V,w)$*-relation*. A *spanner* is a function $P$ that maps every $w \in \Sigma^*$ to a $(V,w)$-relation $P(w)$. Let $V$ be denoted by $\mathsf{SVars}\,(P)$. Two spanners $P_1$ and $P_2$ are equivalent if $\mathsf{SVars}\,(P_1) = \mathsf{SVars}\,(P_2)$, and $P_1(w) = P_2(w)$ for every $w \in \Sigma^*$.

Hence, a spanner can be understood as a function that maps a word $w$ to a set of functions, each of which assigns spans of $w$ to the variables of the spanner. We now examine a formalism that can be used to define spanners:

▶ **Definition 2.** A *regex formula* is an extension of regular expressions to include variables. The syntax is specified with the recursive rules $\alpha := \emptyset \mid \varepsilon \mid a \mid (\alpha \vee \alpha) \mid (\alpha \cdot \alpha) \mid (\alpha)^* \mid x\{\alpha\}$ for $a \in \Sigma$, $x \in \Xi$. We add and omit parentheses freely, as long as the meaning remains clear, and use $\alpha^+$ as shorthand for $\alpha \cdot \alpha^*$, and $\Sigma$ as shorthand for $\bigvee_{a \in \Sigma} a$.

Regex formulas can be interpreted as special case of so-called *regex*, which extend classical regular expressions with a repetition operator (see Section 4.3 for a brief and [14] for a more detailed discussion). This applies to syntax and semantics. In particular, both models define their syntax with parse trees, which is rather impractical for many of our proofs. Instead of using this definition, we present one that is based on *ref-words* (short for *reference words*) by Schmid [25]. A ref-word is a word over the extended alphabet $(\Sigma \cup \Gamma)$, where $\Gamma := \{\vdash_x, \dashv_x \mid x \in \Xi\}$. Intuitively, the symbols $\vdash_x$ and $\dashv_x$ mark the beginning and the end of the span that belongs to the variable $x$. In order to define the semantics of regex formulas, we treat them as generators of ref-languages (i. e., languages of ref-words):

▶ **Definition 3.** For every regex formula $\alpha$, we define its *ref-language* $\mathcal{R}(\alpha)$ by $\mathcal{R}(\emptyset) := \emptyset$, $\mathcal{R}(a) := \{a\}$ for $a \in \Sigma \cup \{\varepsilon\}$, $\mathcal{R}(\alpha_1 \vee \alpha_2) := \mathcal{R}(\alpha_1) \cup \mathcal{R}(\alpha_2)$, $\mathcal{R}(\alpha_1 \cdot \alpha_2) := \mathcal{R}(\alpha_1) \cdot \mathcal{R}(\alpha_2)$, $\mathcal{R}(\alpha_1^*) := \mathcal{R}(\alpha_1)^*$, and $\mathcal{R}(x\{\alpha_1\}) := \vdash_x \mathcal{R}(\alpha_1) \dashv_x$.

Let $\mathsf{SVars}\,(\alpha)$ be the set of all $x \in \Xi$ such that $x\{\,\}$ occurs in $\alpha$. A ref-word $r \in \mathcal{R}(\alpha)$ is *valid* if, for every $x \in \mathsf{SVars}\,(\alpha)$, $|r|_{\vdash_x} = 1$. Let $\mathsf{Ref}(\alpha) := \{r \in \mathcal{R}(\alpha) \mid r \text{ is valid}\}$. We call $\alpha$ *functional* if $\mathsf{Ref}(\alpha) = \mathcal{R}(\alpha)$, and denote the set of all functional regex formulas by $\mathsf{RGX}$.

In other words, $\mathcal{R}(\alpha)$ treats $\alpha$ like a standard regular expression over the alphabet $(\Sigma \cup \Gamma)$, where $x\{\alpha_1\}$ is interpreted as $\vdash_x \alpha_1 \dashv_x$. Furthermore, $\mathsf{Ref}(\alpha)$ contains exactly those words where each variable $x$ is opened and closed exactly once.

▶ **Example 4.** Define regex formulas $\alpha := (x\{\mathtt{a}\}y\{\mathtt{b}\}) \vee (y\{\mathtt{a}\}x\{\mathtt{b}\})$, $\beta := x\{\mathtt{a}\} \vee y\{\mathtt{a}\}$, and $\gamma := x\{a\}x\{a\}$. Then $\alpha$ is a functional, while $\beta$ and $\gamma$ are not (in fact, $\mathsf{Ref}(\alpha) = \mathsf{Ref}(\beta) = \emptyset$).

Like [12, 14], this paper only examines functional regex formulas. Hence, without loss of generality, we assume that no variable binding $x\{\,\}$ occurs under a Kleene star $*$.

The definition of $\mathcal{R}(\alpha)$ implies that every $r \in \mathsf{Ref}(\alpha)$ has a unique factorization $r = r_1 \vdash_x r_2 \dashv_x r_3$ for every $x \in \mathsf{SVars}\,(\alpha)$. This can be used to define $\mu(x)$ (i. e., the span that is assigned to $x$). To this purpose, we define a morphism $\mathsf{clr}\colon (\Sigma \cup \Gamma)^* \to \Sigma^*$ by $\mathsf{clr}(a) := a$ for all $a \in \Sigma$, and $\mathsf{clr}(g) := \varepsilon$ for all $g \in \Gamma$ (in other words, $\mathsf{clr}$ projects ref-words to $\Sigma$). Then $\mathsf{clr}(r_1)$ contains the part of $w$ that precedes $\mu(x)$, and $\mathsf{clr}(r_2)$ contains $w_{\mu(x)}$.

For $\alpha \in \mathsf{RGX}$ and $w \in \Sigma^*$, let $\mathsf{Ref}(\alpha, w) := \{r \in \mathsf{Ref}(\alpha) \mid \mathsf{clr}(r) = w\}$. Then every word of $\mathsf{Ref}(\alpha, w)$ encodes one possibility of assigning variables in $w$ that is consistent with $\alpha$.

▶ **Definition 5.** Let $\alpha \in \mathsf{RGX}$, $w \in \Sigma^*$, and $V := \mathsf{SVars}\,(\alpha)$. Every $r \in \mathsf{Ref}(\alpha, w)$ defines a $(V, w)$-tuple $\mu^r$ in the following way: For every $x \in \mathsf{Vars}\,(\alpha)$, there exist uniquely defined $r_1, r_2, r_3$ with $r = r_1 \vdash_x r_2 \dashv_x r_3$. Then $\mu^r(x) := [|\mathsf{clr}(r_1)| + 1, |\mathsf{clr}(r_1 r_2)| + 1\rangle$. The function $[\![\alpha]\!]$ from words $w \in \Sigma^*$ to $(V, w)$-relations is defined by $[\![\alpha]\!](w) := \{\mu^r \mid r \in \mathsf{Ref}(\alpha, w)\}$.

▶ **Example 6.** Assume that $\mathtt{a}, \mathtt{b} \in \Sigma$. We define the functional regex formula

$$\alpha := \Sigma^* \cdot x\big\{\mathtt{a} \cdot y\{\Sigma^*\} \cdot (z\{\mathtt{a}\} \vee z\{\mathtt{b}\})\big\} \cdot \Sigma^*.$$

Let $w := \mathtt{baaba}$. Then $[\![\alpha]\!](w)$ consists of the tuples $([2,4\rangle, [3,3\rangle, [3,4\rangle)$, $([2,5\rangle, [3,4\rangle, [4,5\rangle)$, $([2,6\rangle, [3,5\rangle, [5,6\rangle)$, $([3,5\rangle, [4,4\rangle, [4,5\rangle)$, $([3,6\rangle, [4,5\rangle, [5,6\rangle)$.

As one example of an $r \in \mathsf{Ref}(\alpha, w)$, consider $r = \mathtt{b}\vdash_x \mathtt{a} \vdash_y \mathtt{a} \dashv_y \vdash_z \mathtt{b} \dashv_z \dashv_x \mathtt{a}$. This yields $\mu^r(x) = [2,5\rangle$, $\mu^r(y) = [3,4\rangle$, and $\mu^r(z) = [4,5\rangle$.

It is easily seen that the definition of $[\![\alpha]\!]$ with ref-words is equivalent to the definition from [12]; and so is the definition of functional regex formulas. Basing the definition of semantics on ref-words has two advantages: Firstly, treating $\mathcal{R}(\alpha)$ as a language over $(\Sigma \cup \Gamma)$ allows us to use standard techniques from automata theory, and secondly, it generalizes well to two automata models for defining spanners from [12]. We begin with the first model:

▶ **Definition 7.** Let $V \subset \Xi$ be a finite set of variables, and define $\Gamma_V := \{\vdash_x, \dashv_x \mid x \in V\}$. A *variable set automaton (vset-automaton)* over $\Sigma$ with variables $V$ is a tuple $A = (Q, q_0, q_f, \delta)$, where $Q$ is the set of states, $q_0, q_f \in Q$ are the initial and the final state, and $\delta\colon Q \times (\Sigma \cup \{\varepsilon\} \cup \Gamma_V) \to 2^Q$ is the transition function.

We interpret $A$ as a directed graph, where the nodes are the elements of $Q$, each $q \in \delta(p, a)$ is represented with an edge from $p$ to $q$ with label $a$, where $p \in Q$ and $a \in (\Sigma \cup \{\varepsilon\} \cup \Gamma_V)$. We extend $\delta$ to $\hat{\delta}\colon Q \times (\Sigma \cup \Gamma_V)^* \to 2^Q$ such that for all $p, q \in Q$ and $r \in (\Sigma \cup \Gamma_V)^*$, $q \in \hat{\delta}(p, r)$ if and only if there is a path from $p$ to $q$ that is labeled with $r$. We use this to define $\mathcal{R}(A) := \{r \in (\Sigma \cup \Gamma_V)^* \mid q_f \in \hat{\delta}(q_0, r)\}$.

Let $\mathsf{SVars}\,(A)$ be the set of all $x \in V$ such that $\vdash_x$ or $\dashv_x$ occurs in $A$. A ref-word $r \in \mathcal{R}(A)$ is *valid* if, for every $x \in \mathsf{SVars}\,(A)$, $|r|_{\vdash_x} = |r|_{\dashv_x} = 1$, and $\vdash_x$ occurs to the left of $\dashv_x$. Then $\mathsf{Ref}(A)$, $\mathsf{Ref}(A, w)$, and $[\![A]\!]$ are defined analogously to regex formulas.

Hence, a vset-automaton can be understood as an NFA over $\Sigma$ that has additional transitions that open and close variables. When using ref-words, it is interpreted as NFA over the alphabet $(\Sigma \cup \Gamma)$, and defines the ref-language $\mathcal{R}(A)$; and $\mathsf{Ref}(A)$ is the subset of $\mathcal{R}(A)$ where each variable in $V$ is opened and closed exactly once (and the two operations occur in the correct order). This also demonstrates why our definition is equivalent to the definition from [12] (there, the condition that every variable has to be opened and closed exactly once is realized by the definition of the successor relation for configurations). In particular, every word in $\mathsf{Ref}(A)$ encodes an accepting run of $A$ (as defined in [12]).

Although interpreting vset-automata as acceptors of ref-languages is often convenient, it comes with a caveat. While $\mathsf{Ref}(A_1) = \mathsf{Ref}(A_2)$ implies $[\![A_1]\!] = [\![A_2]\!]$ for all $A_1, A_2 \in \mathsf{VA}_{\mathsf{set}}$, the

converse does not hold: Consider the two ref-words $r_1 := \vdash_x \vdash_y \mathsf{a} \dashv_y \dashv_x$ and $r_2 := \vdash_y \vdash_x \mathsf{a} \dashv_x \dashv_y$. Both define the same $\mathsf{a}$-tuple $\mu$ (with $\mu(x) = \mu(y) = [1,2\rangle$), although $r_1 \neq r_2$.

Fagin et al. [12] also introduced the *variable stack automaton (vstk-automaton)*. Its definition is almost identical to vset-automata, the only difference is that instead of using a distinct symbol $\dashv_x$ for every variable $x$, vstk-automata have only a single closing symbol $\dashv$, which closes the variable that was opened most recently (hence the "stack" in "variable stack automaton"). From now on, assume that $\Gamma$ also includes $\dashv$, and extend $\mathsf{clr}$ by defining $\mathsf{clr}(\dashv) := \varepsilon$. For every vstk-automaton $A$, $\mathcal{R}(A)$ and $\mathsf{SVars}(A)$ are defined as for vset-automata. We define $\mathsf{Ref}(A)$ as the set of all valid $r \in \mathcal{R}(A)$, where $r$ is valid if, for each $x \in \mathsf{SVars}(A)$, $\vdash_x$ occurs exactly once in $w$, and is closed by a matching $\dashv$. More formally, $r$ is valid if $|r|_{\dashv} = \sum_{x \in \mathsf{SVars}(A)} |r|_{\vdash_x}$, and for every $x \in \mathsf{SVars}(A)$, we have that $|r|_{\vdash_x} = 1$ and $r$ can be uniquely factorized into $r = r_1 \vdash_x r_2 \dashv r_3$, with $|r_2|_{\dashv} = \sum_{x \in \mathsf{SVars}(A)} |r_2|_{\vdash_x}$. This unique factorization allows us to interpret every $r \in \mathsf{Ref}(A)$ as a $\mu^r$ analogously to vset-automata.

We use $\mathsf{VA}_{\mathsf{set}}$ and $\mathsf{VA}_{\mathsf{stk}}$ to denote the set of all vset-automata and all vstk-automata, respectively. We define $\mathsf{VA} := \mathsf{VA}_{\mathsf{set}} \cup \mathsf{VA}_{\mathsf{stk}}$, and refer to the elements of $\mathsf{VA}$ as *v-automata*. An example for each type of v-automata can be found in Figure 1.

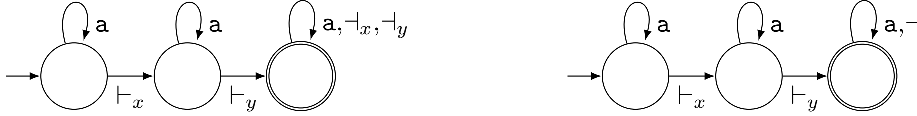



**Figure 1** A vset-automaton $A_{\mathsf{set}}$ (left) and a vstk-automaton $A_{\mathsf{stk}}$ (right). Then $\mathsf{Ref}(A_{\mathsf{set}})$ consist of ref-words $r = \mathsf{a}^{i_1} \vdash_x \mathsf{a}^{i_2} \vdash_y \mathsf{a}^{i_3} \dashv_{z_1} \mathsf{a}^{i_4} \dashv_{z_2} \mathsf{a}^{i_5}$, with $i_1, \ldots, i_5 \geq 0$, $z_1, z_2 \in \{x, y\}$ and $z_1 \neq z_2$. Similarly, the ref-words from $\mathsf{Ref}(A_{\mathsf{stk}})$ are of the form $r = \mathsf{a}^{i_1} \vdash_x \mathsf{a}^{i_2} \vdash_y \mathsf{a}^{i_3} \dashv \mathsf{a}^{i_4} \dashv \mathsf{a}^{i_5}$, with $i_1, \ldots, i_5 \geq 0$. The left $\dashv$ closes $y$, and the right $\dashv$ closes $x$.

### 2.1.2 Spanner Algebras

In order to construct more sophisticated spanners, we introduce spanner operators.

▶ **Definition 8.** Let $P, P_1, P_2$ be spanners. The algebraic operators *union*, *projection*, *natural join* and *selection* are defined as follows.

**Union** $P_1$ and $P_2$ are *union compatible* if $\mathsf{SVars}(P_1) = \mathsf{SVars}(P_2)$, and their *union* $(P_1 \cup P_2)$ is defined by $\mathsf{SVars}(P_1 \cup P_2) := \mathsf{SVars}(P_1)$ and $(P_1 \cup P_2)(w) := P_1(w) \cup P_2(w)$, $w \in \Sigma^*$.

**Projection** Let $Y \subseteq \mathsf{SVars}(P)$. The *projection* $\pi_Y P$ is defined by $\mathsf{SVars}(\pi_Y P) := Y$ and $\pi_Y P(w) := P|_Y(w)$ for all $w \in \Sigma^*$, where $P|_Y(w)$ is the restriction of all $\mu \in P(w)$ to $Y$.

**Natural join** Let $V_i := \mathsf{SVars}(P_i)$ for $i \in \{1, 2\}$. The *(natural) join* $(P_1 \bowtie P_2)$ of $P_1$ and $P_2$ is defined by $\mathsf{SVars}(P_1 \bowtie P_2) := \mathsf{SVars}(P_1) \cup \mathsf{SVars}(P_2)$ and, for all $w \in \Sigma^*$, $(P_1 \bowtie P_2)(w)$ is the set of all $(V_1 \cup V_2, w)$-tuples $\mu$ for which there exist $\mu_1 \in P_1(w)$ and $\mu_2 \in P_2(w)$ with $\mu|_{V_1}(w) = \mu_1(w)$ and $\mu|_{V_2}(w) = \mu_2(w)$.

**Selection** Let $R \in (\Sigma^*)^k$ be a $k$-ary relation over $\Sigma^*$. The *selection operator* $\zeta^R$ is parameterized by $k$ variables $x_1, \ldots, x_k \in \mathsf{SVars}(P)$, written as $\zeta^R_{x_1,\ldots,x_k}$. The *selection* $\zeta^R_{x_1,\ldots,x_k} P$ is defined by $\mathsf{SVars}\left(\zeta^R_{x_1,\ldots,x_k} P\right) := \mathsf{SVars}(P)$ and, for all $w \in \Sigma^*$, $\zeta^R_{x_1,\ldots,x_k} P(w)$ is the set of all $\mu \in P(w)$ for which $\left(w_{\mu(x_1)}, \ldots, w_{\mu(x_k)}\right) \in R$.

Note that join operates on spans, while selection operates on the subwords of $w$ that are described by the spans. Like [12] (also see the brief remark on core spanners below), we mostly consider the string equality selection operator $\zeta^=$ . Hence, unless otherwise noted, the term "selection" refers to selection by the $k$-ary string equality relation. Regarding

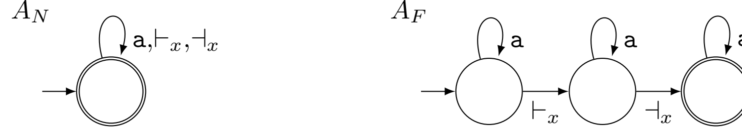



**Figure 2** Twot vset-automata $A_N$ and $A_F$, which both define the universal spanner for the single variable $x$ (cf. [12]) over the alphabet $\{\mathtt{a}\}$. As $\mathcal{R}(A_N)$ contains ref-words like $\mathtt{a}\dashv_x\mathtt{a}\vdash_x$ or $\mathtt{a}\vdash_x\mathtt{a}\vdash_x$, $A_N$ is not functional. In contrast to this, $A_F$ is functional, as it uses its three states to ensure that its ref-words contain each of $\vdash_x$ and $\dashv_x$ exactly once, and in the right order.

the join of two spanners $P_1$ and $P_2$, $P_1 \bowtie P_2$ is equivalent to the intersection $P_1 \cap P_2$ if $\mathsf{SVars}\,(P_1) = \mathsf{SVars}\,(P_2)$, and to the Cartesian Product $P_1 \times P_2$ if $\mathsf{SVars}\,(P_1)$ and $\mathsf{SVars}\,(P_2)$ are disjoint. If applicable, we write $\cap$ and $\times$ instead of $\bowtie$.

We refer to regex formulas and v-automata as *primitive spanner representations*. A *spanner algebra* is a finite set of spanner operators. If $\mathsf{O}$ is a spanner algebra and $C$ is a class of primitive spanner representations, then $C^{\mathsf{O}}$ denotes the set of all *spanner representations* that can be constructed by (repeated) combination of the symbols for the operators from $\mathsf{O}$ with regex formulas from $C$. For each spanner representation of the form $o\rho$ (or $\rho_1\; o\; \rho_2$), where $o \in \mathsf{O}$, we define $[\![o\rho]\!] = o[\![\rho]\!]$ (and $[\![\rho_1\; o\; \rho_2]\!] = [\![\rho_1]\!]\; o\; [\![\rho_2]\!]$). Furthermore, $[\![C^{\mathsf{O}}]\!]$ is the closure of $[\![C]\!]$ under the spanner operators in $\mathsf{O}$.

Fagin et al. [12] refer to $[\![\mathsf{RGX}^{\{\pi,\zeta^=,\cup,\bowtie\}}]\!]$ as the class of *core spanners*, as these capture the core of the functionality of SystemT. Following this, we define $\mathsf{core} := \{\pi, \zeta^=, \cup, \bowtie\}$. This allows us to use more compact notation, like $\mathsf{RGX}^{\mathsf{core}}$, $\mathsf{VA}_{\mathsf{set}}^{\mathsf{core}}$, $\mathsf{VA}_{\mathsf{stk}}^{\mathsf{core}}$, and $\mathsf{VA}^{\mathsf{core}}$.

### 2.1.3 Some Results on Automata-Based Spanners

This section develops some basic insights on aspects of v-automata, which we later use to provide further context to the main result in Section 3. While [12] defines $\mathsf{RGX}$ as the set of functional regex formulas, no analogous restriction is used for $\mathsf{VA}_{\mathsf{set}}$ and $\mathsf{VA}_{\mathsf{stk}}$. Using ref-word terminology, this means that for each $\alpha \in \mathsf{RGX}$, all information that is needed to determine $\mathsf{Ref}(\alpha, w)$ can be derived from $\mathcal{R}(\alpha)$. We adapt this notion to v-automata, and call $A \in \mathsf{VA}$ *functional* if $\mathsf{Ref}(A) = \mathcal{R}(A)$. Figure 2 contains examples for (non-)functional vset-automata (similar observations can be made for vstk-automata). This definition is also natural under the semantics as defined in [12]: Translated to these semantics, a v-automaton $A$ is functional if every path from $q_0$ to $q_f$ yields an accepting run of $A$. While v-automata in general have to keep track of the used variables, functional v-automata store this information implicitly in their states. Hence, their evaluation problem can be solved efficiently:

▶ **Lemma 9.** *Given $w \in \Sigma^*$, a functional $A \in \mathsf{VA}$, and a $(\mathsf{SVars}\,(A), w)$-tuple $\mu$, $\mu \in [\![A]\!](w)$ can be decided in polynomial time.*

With a slight modification of standard reachability techniques, we can show the following:

▶ **Proposition 10.** *Given $A \in \mathsf{VA}$, we can decide in polynomial time whether $A$ is functional.*

In contrast to Lemma 9, even special cases of evaluating non-functional v-automata are hard:

▶ **Lemma 11.** *Given $A \in \mathsf{VA}$, deciding whether $[\![A]\!](\varepsilon) \neq \emptyset$ is $\mathsf{NP}$-complete.*

The proof uses a basic reduction from the Hamiltonian path problem, which is $\mathsf{NP}$-complete (cf. Garey and Johnson [15]). We discuss the matching upper bound in Section 3.

Obviously, every vset- or vstk-automaton can be transformed into an equivalent functional automaton, by intersecting with an NFA that accepts the set of all valid ref-words, using the standard constructions for NFA-intersection. Lemma 11 already suggests that this conversion is not possible in polynomial time (unless the number of variables is bounded); we also show matching exponential size bounds:

▶ **Proposition 12.** *Let $f_{\mathsf{set}}(k) := 3^k$, $f_{\mathsf{stk}}(k) := (k+2)2^{k-1}$, and $s \in \{\mathsf{set}, \mathsf{stk}\}$. For every $A \in \mathsf{VA}_s$ with $n$ states and $k$ variables, there exists an equivalent functional $A_F \in \mathsf{VA}_s$ with $n \cdot f_s(k)$ states. For every $k \geq 1$, there is an $A_k \in \mathsf{VA}_s$ with one state and $k$ variables, such that every equivalent functional $A_F \in \mathsf{VA}_s$ has at least $f_s(k)$ states.*

The lower bounds are obtained by treating the v-automata as NFAs, which allows the use of a fooling set technique by Birget [2]. We briefly compare vset- and vstk-automata: As shown in [12], $[\![\mathsf{VA}_{\mathsf{stk}}]\!] \subset [\![\mathsf{VA}_{\mathsf{set}}]\!]$. The reason for this is that, as vstk-automata always close the variable that was opened most recently, they can only express hierarchical spanners (a spanner is hierarchical if its spans do not overlap – for a formal definition, see [12]). While this behavior can be simulated with vset-automata, a slight modification of the proof of Proposition 12 shows that this is not possible in an efficient manner:

▶ **Proposition 13.** *For every $k \geq 1$, there is a vstk-automaton $A_k$ with one state and $k+2$ edges, such that every vset-automaton $A$ with $[\![A]\!] = [\![A_k]\!]$ has at least $k!$ states.*

Hence, although vstk-automata can express strictly less than vset-automata, they may offer an exponential succinctness advantage. We revisit this in Section 3.

## 2.2 Word Equations and $\mathsf{EC}^{\mathsf{reg}}$

A *pattern* is a word $\alpha \in (\Sigma \cup \Xi)^*$, and a *word equation* is a pair of patterns $(\eta_L, \eta_R)$, which can also be written as $\eta_L = \eta_R$. A *pattern substitution* (or just *substitution)* is a morphism $\sigma\colon (\Xi \cup \Sigma)^* \to \Sigma^*$ with $\sigma(a) = a$ for all $a \in \Sigma$. Recall that a morphism from a free monoid $A^*$ to a free monoid $B^*$ is a function $h\colon A^* \to B^*$ such that $h(x \cdot y) = h(x) \cdot h(y)$ for all $x, y \in A^*$. Hence, in order to define $h$, it suffices to define $h(x)$ for all $x \in A$. Therefore, we can uniquely define a pattern substitution $\sigma$ by defining $\sigma(x)$ for each $x \in \Xi$.

A substitution $\sigma$ is a *solution* of a word equation $(\eta_L, \eta_R)$ if $\sigma(\eta_L) = \sigma(\eta_R)$. The set of all variables in a pattern $\alpha$ is denoted by $\mathsf{var}(\alpha)$. We extend this to word equations $\eta = (\eta_L, \eta_R)$ by $\mathsf{var}(\eta) := \mathsf{var}(\eta_L) \cup \mathsf{var}(\eta_R)$.

The *existential theory of concatenation* $\mathsf{EC}$ is obtained by combining word equations with $\wedge$, $\vee$, and existential quantification over variables. Formally, every word equation $\eta$ is an $\mathsf{EC}$-formula, and $\sigma \models \eta$ if $\sigma$ is a solution of $\eta$. If $\varphi_1$ and $\varphi_2$ are $\mathsf{EC}$-formulas, so are $\varphi_\wedge := (\varphi_1 \wedge \varphi_2)$ and $\varphi_\vee := (\varphi_1 \vee \varphi_2)$, with $\sigma \models \varphi_\wedge$ if $\sigma \models \varphi_1$ and $\sigma \models \varphi_2$; and $\sigma \models \varphi_\vee$ if $\sigma \models \varphi_1$ or $\sigma \models \varphi_2$. Finally, for every $\mathsf{EC}$-formula $\varphi$ and every $x \in \Xi$, $\psi := (\exists x\colon \varphi)$ is an $\mathsf{EC}$-formula, and $\sigma \models \psi$ if there exists a $w \in \Sigma^*$ such that $\sigma_{[x \to w]} \models \varphi$, where the substitution $\sigma_{[x \to w]}$ is defined by $\sigma_{[x \to w]}(y) := w$ if $y = x$, and $\sigma_{[x \to w]}(y) := \sigma(y)$ if $y \neq x$.

We also consider the *existential theory of concatenation with regular constraints*, $\mathsf{EC}^{\mathsf{reg}}$. In addition to word equations, $\mathsf{EC}^{\mathsf{reg}}$-formulas can use constraints $\mathsf{C}_A(x)$, where $x \in \Xi$ is a variable, $A$ is an NFA, and $\sigma \models \mathsf{C}_A(x)$ if $\sigma(x) \in \mathcal{L}(A)$. As every regular expression can be directly converted into an equivalent NFA, we also allow constraints $\mathsf{C}_\alpha(x)$ that use regular expressions instead of NFAs. We freely omit parentheses, as long as the meaning of the formula remains unambiguous. To increase readability, we allow existential quantifiers to range over multiple variables; i. e., we use $\exists x_1, x_2, \ldots, x_k\colon \varphi$ as a shorthand for $\exists x_1\colon \exists x_2\colon \ldots \exists x_k\colon \varphi$.

The set $\mathsf{free}(\varphi)$ of *free variables* of an $\mathsf{EC}^{\mathsf{reg}}$-formula $\varphi$ is defined by $\mathsf{free}(\eta) = \mathsf{var}(\eta)$, $\mathsf{free}(\varphi_1 \wedge \varphi_2) := \mathsf{free}(\varphi_1 \vee \varphi_2) := \mathsf{free}(\varphi_1) \cup \mathsf{free}(\varphi_2)$, and $\mathsf{free}(\exists x \colon \varphi) := \mathsf{free}(\varphi) - \{x\}$. Finally, we define $\mathsf{free}(C) = \emptyset$ for every constraint $C$. (While one could also argue in favor of $\mathsf{free}(C(x)) = \{x\}$, choosing $\emptyset$ simplifies the definitions in Section 3). For all $\varphi \in \mathsf{EC}^{\mathsf{reg}}$, let $[\![\varphi]\!] := \{\sigma \mid \sigma \models \varphi\}$. Two formulas $\varphi_1, \varphi_2 \in \mathsf{EC}^{\mathsf{reg}}$ are *equivalent* if $\mathsf{free}(\varphi_1) = \mathsf{free}(\varphi_2)$ and $[\![\varphi_1]\!] = [\![\varphi_2]\!]$. We write this as $\varphi_1 \equiv \varphi_2$. For increased readability, we use $\varphi(x_1, \ldots, x_k)$ to denote $\mathsf{free}(\varphi) = \{x_1, \ldots, x_k\}$. Building on this, we also use $(w_1, \ldots, w_k) \models \varphi(x_1, \ldots, x_k)$ to denote $\sigma \models \varphi$ for the substitution $\sigma$ that is defined by $\sigma(x_i) := w_i$, $1 \leq i \leq k$.

▶ **Example 14.** Consider the $\mathsf{EC}$-formula $\varphi_1(x, y, z) := \exists \hat{x}, \hat{y} \colon (x = z\hat{x} \wedge y = z\hat{y})$ and the $\mathsf{EC}^{\mathsf{reg}}$-formula $\varphi_1(x, y, z) := \exists \hat{x}, \hat{y} \colon (x = z\hat{x} \wedge y = z\hat{y} \wedge \mathsf{C}_{\Sigma^+}(z))$ . Then $\sigma \models \varphi_1$ if and only if $\sigma(x)$ and $\sigma(y)$ have $\sigma(z)$ as common prefix. If, in addition to this, $\sigma(z) \neq \varepsilon$, then $\sigma \models \varphi_2$.

Every $\mathsf{EC}$-formula can be converted into a single word equation (cf. Karhumäki, Mignosi, and Plandowski [20]), and every $\mathsf{EC}^{\mathsf{reg}}$-formula into a single word equation with rational constraints (cf. Diekert [7]). For conjunctions, the construction is easily explained: Choose distinct letters $\mathtt{a}, \mathtt{b} \in \Sigma$. Hmelevskii's pattern pairing function is defined by $\langle \alpha, \beta \rangle := \alpha \mathtt{a} \beta \mathtt{a} \alpha \mathtt{b} \beta \mathtt{b}$. Then $(\alpha_L = \alpha_R) \wedge (\beta_L = \beta_R)$ holds if and only if $\langle \alpha_L, \beta_L \rangle = \langle \alpha_R, \beta_R \rangle$. The construction for disjunctions is similar, but more involved (and, in general, converting a formula with alternating disjunctions and conjunctions leads to an exponential size increase).

Satisfiability for $\mathsf{EC}^{\mathsf{reg}}$ is $\mathsf{PSPACE}$-complete; but even for $\mathsf{EC}$, showing the upper bound is by no means trivial (cf. [7, 9]). Note that negation is left out intentionally: Even the $\mathsf{EC}$-fragment $\forall\exists^3$ (one universal over three existential variables) is undecidable (Durnev [10]).

## 3 SpLog: A Logic for Spanners

As shown by Freydenberger and Holldack [14], every element of $\mathsf{RGX}^{\mathsf{core}}$ can be converted into an $\mathsf{EC}^{\mathsf{reg}}$-formula, and every word equation with regular constraints (and, hence, every $\mathsf{EC}^{\mathsf{reg}}$-formula) can be converted to $\mathsf{RGX}^{\mathsf{core}}$. While the latter results in a spanner that is satisfiable if and only if the formula is satisfiable, the input word of the spanner needs to encode the whole word equation (see the comments after Example 14). Hence, the spanner can only simulate satisfiability, but not evaluation. To overcome this problem, we introduce $\mathsf{SpLog}$ (short for *spanner logic*), a fragment of $\mathsf{EC}^{\mathsf{reg}}$ that directly corresponds to core spanners:

▶ **Definition 15.** A formula $\varphi \in \mathsf{EC}$ is called *safe* if the following two conditions are met:

1. If $(\varphi_1 \vee \varphi_2)$ is a subformula of $\varphi$, then $\mathsf{free}(\varphi_1) = \mathsf{free}(\varphi_2)$.
2. Every constraint $\mathsf{C}_A(x)$ occurs only as part of a subformula $(\psi \wedge \mathsf{C}_A(x))$, with $x \in \mathsf{free}(\psi)$.

Let $W \in \Xi$. The set of all $\mathsf{SpLog}$-*formulas with main variable* $\mathsf{W}$, $\mathsf{SpLog}(\mathsf{W})$, is the set of all safe $\varphi \in \mathsf{EC}^{\mathsf{reg}}$ such that

1. all word equations in $\varphi$ are of the form $\mathsf{W} = \eta_R$, with $\eta_R \in ((\Xi - \{\mathsf{W}\}) \cup \Sigma)^*$,
2. for every subformula $\psi$ of $\varphi$, $\mathsf{W} \in \mathsf{free}(\psi)$.

We also define the set of all $\mathsf{SpLog}$-formulas by $\mathsf{SpLog} := \bigcup_{W \in \Xi} \mathsf{SpLog}(\mathsf{W})$, and we use $\mathsf{SpLog}_{\mathsf{rx}}$ to denote the fragment of $\mathsf{SpLog}$ that exclusively defines constraints with regular expressions instead of NFAs.

Less formally, for every $\varphi \in \mathsf{SpLog}(\mathsf{W})$, the main variable $\mathsf{W}$ appears on the left side of every equation (and is never bound with a quantifier). The requirement that $\varphi$ is safe ensures that each variable corresponds to a subword of $\mathsf{W}$. When declaring the free variables of a $\mathsf{SpLog}$-formula, we slightly diverge from our convention for $\mathsf{EC}^{\mathsf{reg}}$-formulas, and write $\varphi(\mathsf{W}; x_1, \ldots, x_k)$ to denote a formula with main variable $\mathsf{W}$, and $\mathsf{free}(\varphi) = \{\mathsf{W}, x_1, \ldots, x_k\}$.

▶ **Example 16.** Let $\varphi_1(\mathsf{W}; x) := \exists y, z_1, z_2 \colon (\mathsf{W} = yy \land \mathsf{W} = z_1 x z_2 \land \mathsf{C}_{\Sigma^+}(x))$. Then $\varphi_1$ is a $\mathsf{SpLog}(\mathsf{W})$-formula, and $\sigma \models \varphi_1$ iff. $\sigma(\mathsf{W})$ is a square and contains $\sigma(x)$ as a nonempty subword. In contrast to this, $\varphi_2(\mathsf{W}; x, y) := (\mathsf{W} = xx \lor \mathsf{W} = yyy)$ is not a SpLog-formula, as it is not safe (intuitively, if e. g. $\sigma(\mathsf{W}) = \sigma(x)^2$, then $\sigma \models \varphi_2$, even if $\sigma(y) \not\sqsubseteq \sigma(\mathsf{W})$). Further examples for SpLog-formulas can be found in Section 4.

Before we examine conversions between SpLog and various representations of core spanners, we introduce a result that provides us with a convenient shorthand notation:

▶ **Lemma 17.** *Let $\varphi \in \mathsf{SpLog}(\mathsf{W})$, $x \in \mathsf{free}(\varphi) - \{\mathsf{W}\}$, and let $\psi \in \mathsf{SpLog}(x)$ such that* W *does not occur in $\psi$. We can compute in polynomial time a $\chi \in \mathsf{SpLog}(\mathsf{W})$ with $\chi \equiv (\varphi \land \psi)$.*

**Proof.** Let $x_1, x_2$ be new variables and define $\chi := \varphi \land \exists x_1, x_2 \colon \big((W = x_1 \cdot x \cdot x_2) \land \hat{\psi}\big)$, where $\hat{\psi}$ is obtained from $\psi$ by replacing every equation $x = \eta_R$ with $W = x_1 \cdot \eta_R \cdot x_2$. Given $W = x_1 \cdot x \cdot x_2$, these equations define the same relations as the $x = \eta_R$. As W does not occur in $\psi$, $\chi \equiv (\varphi \land \psi)$ holds. ◀

This allows us to combine SpLog-formulas with different main variables.

When comparing the expressive power of spanners and SpLog, we need to address one important difference of the two models: While SpLog is defined on words, spanners are defined on spans of an input word. Apart from slight modifications to adapt it to SpLog, the following definition for the conversion of spanners to formulas was introduced in [14]:

▶ **Definition 18.** Let $P$ be a spanner and let $\varphi \in \mathsf{SpLog}(\mathsf{W})$ with $\mathsf{free}(\varphi) = \{\mathsf{W}\} \cup \{x^P, x^C \mid x \in \mathsf{SVars}(P)\}$. We say that $\varphi$ *realizes* $P$ if, for all substitutions $\sigma$, $\sigma \models \varphi$ holds if and only if there is a $\mu \in P(\sigma(\mathsf{W}))$ such that, for each $x \in \mathsf{SVars}(P)$, $\sigma(x^P) = \sigma(\mathsf{W})_{[1,i\rangle}$ and $\sigma(x^C) = \sigma(\mathsf{W})_{[i,j\rangle}$, where $[i, j\rangle = \mu(x)$.

The intuition behind this definition is that every span $[i, j\rangle$ of $w$ is characterized by its content $w_{[i,j\rangle}$, and by $w_{[1,i\rangle}$, the prefix of $w$ that precedes the span. Hence, every variable $x$ of the spanner is represented by two variables $x^C$ and $x^P$, which store the content and the prefix, respectively. Moreover, the main variable of the SpLog-formula corresponds to the input word of the spanner. Next, we consider conversions in the other direction:

▶ **Definition 19.** Let $\varphi \in \mathsf{SpLog}(\mathsf{W})$. A spanner $P$ with $\mathsf{SVars}(P) = \mathsf{free}(\varphi) - \{\mathsf{W}\}$ *realizes* $\varphi$ if, for all substitutions $\sigma$, $\sigma \models \varphi$ holds if and only if there is a $\mu \in P(\sigma(\mathsf{W}))$ such that $\sigma(\mathsf{W})_{\mu(x)} = \sigma(x)$ for all $x \in \mathsf{SVars}(P)$.

Again, the main variable of the SpLog-formula corresponds to the input word of the spanner. Note that it is possible to define realizability in a stricter way: Instead of requiring that $\mu \in P(\sigma(\mathsf{W}))$ holds for *one* $\mu$ with $\sigma(\mathsf{W})_{\mu(x)} = \sigma(x)$ for all $x \in \mathsf{SVars}(P)$, we could require $\mu \in P(\sigma(\mathsf{W}))$ for *all such* $\mu$. But such a spanner can directly be constructed from a spanner $P$ that satisfies Definition 19, by joining $P$ with a universal spanner (cf. [12]), and using string equality selections (for the matter of this paper, this will not affect the complexity, as consider spanners with string equality relations).´

Let $C_1$ be a class of spanner representations (or SpLog-formulas), and let $C_2$ be a class of SpLog-formulas (or spanner representations). We say that there is a *polynomial size conversion* from $C_1$ to $C_2$ if there is an algorithm that, given a $\rho_1 \in C_1$, computes a $\rho_2 \in C_2$ such that $\rho_2$ realizes $\rho_1$, and the size of $\rho_2$ is polynomial in the size of $\rho_1$. If the algorithm also works in polynomial time, we say that there is a *polynomial time conversion*. First, we use Lemma 11 to obtain a negative result on conversions to SpLog:

▶ **Lemma 20.** $\mathsf{P} = \mathsf{NP}$*, if there is a polynomial time conversion from* $\mathsf{VA}_{\mathsf{set}}$ *or* $\mathsf{VA}_{\mathsf{stk}}$ *to* SpLog*.*

This result is less problematic than it might appear, as it can be overcome with a very minor relaxation of the definition of polynomial time conversions: We say that a SpLog-formula $\varphi$ *realizes a spanner $P$ modulo $\varepsilon$* if $\varphi$ realizes a spanner $\hat{P}$ with $P(w) = \hat{P}(w)$ for all $w \in \Sigma^+$. In other words, $\varphi$ realizes $P$ on all inputs, except $\varepsilon$ (where the behavior is undefined). Likewise, a *polynomial time conversion modulo $\varepsilon$* computes formulas that realize the spanners modulo $\varepsilon$. We now state the central result of this paper:

▶ **Theorem 21.** *There are polynomial time conversions*
1. *from* $\mathsf{RGX}^{\mathsf{core}}$ *to* $\mathsf{SpLog}_{\mathsf{rx}}$*, and from* $\mathsf{SpLog}_{\mathsf{rx}}$ *to* $\mathsf{RGX}^{\mathsf{core}}$*,*
2. *from* SpLog *to* $\mathsf{VA}^{\mathsf{core}}_{\mathsf{set}}$ *and to* $\mathsf{VA}^{\mathsf{core}}_{\mathsf{stk}}$*,*
3. *modulo $\varepsilon$ from* $\mathsf{VA}^{\mathsf{core}}_{\mathsf{set}}$ *and* $\mathsf{VA}^{\mathsf{core}}_{\mathsf{stk}}$ *to* SpLog*.*

Recall that $\mathsf{SpLog}_{\mathsf{rx}}$ is the fragment of SpLog that uses only regular expressions to define constraints. The conversion from $\mathsf{RGX}^{\mathsf{core}}$ to $\mathsf{SpLog}_{\mathsf{rx}}$ is almost identical to the conversion from $\mathsf{RGX}^{\mathsf{core}}$ to $\mathsf{EC}^{\mathsf{reg}}$ that was presented in [14]. The most technically challenging part is the conversion of non-functional v-automata to SpLog, which requires a gadget that acts as a synchronization mechanism inside the formula. This is realized by sets of variables that map to either $\varepsilon$ or the first letter of W, which is the main reason that the construction only works modulo $\varepsilon$. For most applications, $P(\varepsilon)$ can be considered a pathological edge case: As $P(w)$ can be understood as searching in $w$, $P(\varepsilon)$ corresponds to a search in $\varepsilon$. But even if we insist on correctness on $\varepsilon$, we are still able to observe polynomial size conversions:

▶ **Corollary 22.** *There are polynomial size conversions from* $\mathsf{VA}^{\mathsf{core}}$ *to* SpLog*.*

As discussed in Section 2.1.3, there are exponential blowups when moving from general to functional v-automata, as well as from vstk- to vset-automata. Another consequence of Theorem 21 is that this does not hold if we extend the automata with the core-algebra:

▶ **Corollary 23.** *Given $\rho \in \mathsf{VA}^{\mathsf{core}}$, we can compute an equivalent $\rho_f \in \mathsf{VA}_{\mathsf{set}}^{\{\pi,\zeta^=,\cup,\times\}}$ or $\rho_f \in \mathsf{VA}_{\mathsf{stk}}^{\{\pi,\zeta^=,\cup,\times\}}$, where $\rho_f$ is of polynomial size and every v-automaton in $\rho_f$ is functional.*

Again, due to Lemma 11, computing an equivalent $\rho_f$ in polynomial time would imply $\mathsf{P} = \mathsf{NP}$; but we can compute in polynomial time a $\rho_f$ that is equivalent modulo $\varepsilon$.

This also demonstrates that $\bowtie$ can be simulated by a combination of $\times$ and $\zeta^=$, in addition to showing that the algebra compensates the aforementioned disadvantages in succinctness. While we leave open whether there are polynomial size conversions from SpLog to $\mathsf{RGX}^{\mathsf{core}}$, or from $\mathsf{VA}^{\mathsf{core}}$ to $\mathsf{SpLog}_{\mathsf{rx}}$ or $\mathsf{RGX}^{\mathsf{core}}$, we observe that, due to Theorem 21, all these questions are equivalent to asking how efficiently $\mathsf{SpLog}_{\mathsf{rx}}$ can simulate NFAs.

Another question that we leave open is whether $[\![\mathsf{SpLog}]\!] = [\![\mathsf{EC}^{\mathsf{reg}}]\!]$ (see Section 4.4). But we are able to state an important difference between the two logics: While evaluation of $\mathsf{EC}^{\mathsf{reg}}$-formulas is PSPACE-hard, this does not hold for SpLog (assuming $\mathsf{NP} \neq \mathsf{PSPACE}$):

▶ **Corollary 24.** *Given $\varphi \in$ SpLog and a substitution $\sigma$, deciding $\sigma \models \varphi$ is NP-complete. For every fixed $\varphi \in$ SpLog, given a substitution $\sigma$, deciding $\sigma \models \varphi$ is in NL.*

Finally, we remark that Theorem 21 also shows that the PSPACE upper bounds of deciding satisfiability and hierarchicality for $\mathsf{RGX}^{\mathsf{core}}$ that were observed in [14] also apply to $\mathsf{VA}^{\mathsf{core}}_{\mathsf{set}}$ and $\mathsf{VA}^{\mathsf{core}}_{\mathsf{stk}}$. The same holds for the uppers bound for combined and data complexity.

# 4 Expressing Relations in SpLog

This section examines how SpLog expresses relations and languages: Section 4.1 lays the formal groundwork by introducing selectability of relations in SpLog (and connecting it to

core spanners), Section 4.2 contains an extended example, Section 4.3 provides an efficient conversion of a subclass of regex to SpLog, and Section 4.4 defines and applies a normal form.

## 4.1 Selectable Relations

One of the topics of Fagin et al. [12] is which relations can be used for selections in core spanners, without increasing the expressive power. This translates to the question which relations can be used in the definition of SpLog-formulas. For $\mathsf{EC}^{\mathsf{reg}}$, this question is simple: If, for any $k$-ary relation $R$, there is an $\mathsf{EC}^{\mathsf{reg}}$-formula $\varphi_R$ such that $\vec{w} \models \varphi_R$ holds if and only if $\vec{w} \in R$, we know that we can use $\varphi_R$ in the construction of $\mathsf{EC}^{\mathsf{reg}}$-formulas. In contrast to this, the special role of the main variable makes the situation a little bit more complicated for SpLog. Fortunately, [12] already introduced an appropriate concept for core spanners, that we can directly translate to SpLog: A $k$-ary word relation $R$ is *selectable by core spanners* if, for every $\rho \in \mathsf{RGX}^{\mathsf{core}}$ and every sequence of variables $\vec{x} = (x_1, \dots, x_k)$ with $x_1, \dots, x_k \in \mathsf{SVars}(\rho)$, the spanner $[\![\zeta^R_{\vec{x}}\rho]\!]$ is expressible in $\mathsf{RGX}^{\mathsf{core}}$.

Analogously, we say that $R$ is *SpLog-selectable* if for every $\varphi \in \mathsf{SpLog}$ and every sequence of variables $\vec{x} = (x_1, \dots, x_k)$ with $x_1, \dots, x_k \in \mathsf{free}(\varphi) - \{\mathsf{W}\}$, there is a $\varphi^R_{\vec{x}} \in \mathsf{SpLog}$ with $\mathsf{free}(\varphi) = \mathsf{free}(\varphi^R_{\vec{x}})$, and $\sigma \models \varphi^R_{\vec{x}}$ if and only if $\sigma \models \varphi$ and $(\sigma(x_1), \dots, \sigma(x_k)) \in R$. Before we consider some examples, we prove that these two definitions are equivalent not only to each other, but also to a more convenient third definition:

▶ **Lemma 25.** *For every relation $R \subseteq (\Sigma^*)^k$, $k \geq 1$, the following conditions are equivalent:*
1. *$R$ is selectable by core spanners,*
2. *$R$ is SpLog-selectable,*
3. *there is a $\varphi(W; x_1, \dots, x_k) \in \mathsf{SpLog}$ with $\sigma \models \varphi$ if and only if $(\sigma(x_1), \dots, \sigma(x_k)) \in R$.*

The equivalence of the two notions of selectability is one of the features of SpLog: When defining core spanners, one can use SpLog to define relations that are used in selections. As the proof is constructive and uses Theorem 21, this does not even affect efficiency. Before we discuss how the equivalent third condition in Lemma 25 can be used to simplify this even further, we consider a short example. As shown by Fagin et al. [12], the relation $\sqsubseteq$ is selectable by core spanners. We reprove this by showing that it is SpLog-selectable:

▶ **Example 26.** The subword relation $R_{\sqsubseteq} := \{(x, y) \mid x \sqsubseteq y\}$ is selected by the SpLog-formula $\varphi_{\sqsubseteq}(\mathsf{W}; x, y) := \exists z_1, z_2, y_1, y_2 \colon ((\mathsf{W} = z_1 y_1 x y_2 z_2) \wedge (\mathsf{W} = z_1 y z_2))$. If this is not immediately clear, note that the formula implies $z_1 y_1 x y_2 z_2 = z_1 y z_2$, which can be reduced to $y_1 x y_2 = y$.

This allows us to use $x \sqsubseteq y$ as a shorthand in SpLog-formulas. We also use $\sqsubseteq$ to address two inconveniences that arise when strictly observing the syntax of SpLog-formulas: Firstly, the need to introduce additional variables that might affect readability (like $z_1, z_2$ in Example 26), and, secondly, the basic form that equations have the main variable $\mathsf{W}$ on the left side. Together with Lemma 17 and the third condition of Lemma 25, the selectability of $\sqsubseteq$ allows us more compact definitions of SpLog-selectable relations: Instead of dealing with a single main variable, we can combine multiple SpLog-functions with different main variables. Hence, when using SpLog to define a relation over a set of variables $V$, we may assume that the formula is of the form $(\bigwedge_{x \in V} x \sqsubseteq \mathsf{W}) \wedge \varphi$, and specify only $\varphi$. When the main variable is clear, we also omit it, as seen in the following examples:

▶ **Example 27.** Using the aforementioned simplifications, we can write the formula from Example 26 as $\varphi_{\sqsubseteq}(x, y) := \exists y_1, y_2 \colon (y = y_1 \cdot x \cdot y_2)$. Similarly, we can select the prefix relation with the formula $\varphi_{pref}(x, y) := \exists z \colon y = xz$. Both are shorthands for SpLog(W)-formulas.

As mentioned above, this allows us to extend the syntax of SpLog with $x \sqsubseteq y$. Other extensions are $x \neq \varepsilon$ and $x \neq y$: For $x \neq \varepsilon$, we can use $\varphi_{\neq\varepsilon}(x) := (x \sqsubseteq \mathsf{W}) \wedge (\mathsf{C}_{\Sigma^+}(x))$. For the more general $x \neq y$, we consider the following SpLog(W)-formula:

$$\varphi_{\neq}(x, y) := \Big(\big(\exists x_2 \colon (x = yx_2) \wedge (x_2 \neq \varepsilon)\big) \vee \big(\exists y_2 \colon (y = xy_2) \wedge (y_2 \neq \varepsilon)\big)\Big) \vee \Big(\bigvee_{\mathtt{a} \in \Sigma} \big(\exists z, x_2, y_2, b \colon (x = z\mathtt{a}x_2) \wedge (y = zby_2) \wedge \mathsf{C}_{\Sigma - \{\mathtt{a}\}}(b)\big)\Big)$$

The core spanner selectability of $\neq$ was already shown in [12], Proposition 5.2. Depending on personal preferences, $\varphi_{\neq}$ might be considered more readable than the spanner in that proof. A similar construction was also used in [20] to show EC-expressibility of $\neq$.

## 4.2 Extended Example: Relations for Approximate Matching

In this section, we examine how SpLog-formulas can be used to express relations of words that are approximately identical. In literature, this is commonly defined by the notion of an edit distance between two words. Following Navarro [23], we consider edit distances that are based on three operations: For words $u, v \in \Sigma^*$, we say that $v$ can be obtained from $u$ with

1. an *insertion*, if $u = u_1 \cdot u_2$ and $v = u_1 \cdot a \cdot u_2$,
2. a *deletion*, if $u = u_1 \cdot a \cdot u_2$ and $v = u_1 \cdot u_2$,
3. a *replacement*, if $u = u_1 \cdot a \cdot u_2$ and $v = u_1 \cdot b \cdot u_2$,

where $u_1, u_2 \in \Sigma^*$, $a, b \in \Sigma$. For every choice of permitted operations, a distance $d(u, v)$ is then defined as the minimal number of operations that is required to obtain $v$ from $u$. One common example is the *Levenshtein-distance* $d_L$ (also called *edit distance*), which uses insertion, deletion, and replacement. The following SpLog-formula demonstrates that, for each $k \geq 1$, the relation of all $(u, v)$ with $d_L(u, v) \leq k$ is SpLog-selectable:

$$\varphi_{L(k)}(\mathsf{W}; x, y) := \exists x_1, \ldots, x_k, y_1, \ldots, y_k, z_0, \ldots, z_k \colon$$
$$(x = z_0 \cdot x_1 \cdot z_1 \cdot x_2 \cdot z_2 \cdots x_k \cdot z_k) \wedge (y = z_0 \cdot y_1 \cdot z_1 \cdot y_2 \cdot z_2 \cdots y_k \cdot z_k) \wedge \bigwedge_{i=1}^{k} \mathsf{C}_\alpha(x_i) \wedge \bigwedge_{i=1}^{k} \mathsf{C}_\beta(y_i),$$

where $\alpha := \beta := (\Sigma \vee \varepsilon)$. Here, an insertion is expressed by assigning $x_i = \varepsilon$ and $y_i \in \Sigma$, a deletion is modeled by $x_i \in \Sigma$ and $y_i = \varepsilon$, and a replacement by $x_i, y_i \in \Sigma$. This case and $x_i = y_i = \varepsilon$ also cover cases where less than $k$ operations are used.

Hence, by changing the constraints, this formula can also be used for the *Hamming distance* (which uses only replacements), and the *episode distance* (which uses only insertions), by defining $\alpha := \beta := \Sigma$, or $\alpha := \varepsilon$ and $\beta := \Sigma$ (respectively).

With some additional effort, we can also express the relation for the *longest common subsequence distance*, which uses only insertions and deletions. Instead of changing $\alpha$ or $\beta$, we need to ensure that for every $i$, $x_i = \varepsilon$ or $y_i = \varepsilon$ holds. We cannot directly write $((x_i = \varepsilon) \vee (y_i = \varepsilon))$, as this is not a safe formula. Instead, we extend the conjunction inside $\varphi_{L(k)}$ with $\bigwedge_{i=1}^{k}\big(((x_i = \varepsilon) \wedge (y_i \sqsubseteq \mathsf{W})) \vee ((y_i = \varepsilon) \wedge (x_i \sqsubseteq \mathsf{W}))\big)$, which is safe and equivalent to $\bigwedge_{i=1}^{k}((x_i = \varepsilon) \vee (y_i = \varepsilon))$. In other words, we use $\sqsubseteq$ to guard the $x_i$ and $y_i$.

## 4.3 Efficient Conversion of vsf-Regex to SpLog

Most modern implementations of regular expressions contain a backreference operator that allows the definition of non-regular languages. This is formalized in *regex* (also called extended regular expressions), which extend regex formulas with variable references $\&x$ for every $x \in \Xi$.

Intuitively, the semantics of $\&x$ can be understood as repeating the last value that was assigned to $x\{\ \}$, assuming that the regex is parsed left to right (for a formal definition that uses parse trees, see Freydenberger and Holldack [14]; for a definition with ref-words, see Schmid [25] or Section A.12 in the Appendix). For example, $x\{\Sigma^*\} \cdot \&x \cdot \&x$ generates the language of all $www$ with $w \in \Sigma^*$.

As shown by Fagin et al. [12], core spanners cannot define all regex languages. But [14] introduces a subclass of regex, the *vstar-free regex* (short: *vsf-regex*). A vsf-regex is a regex that does not use $x\{\ \}$ or $\&x$ inside a Kleene star $^*$. Every vsf-regex can be converted effectively into a core spanner; but the conversion from [14] can lead to an exponential blowup. The question whether a more efficient conversion is possible was left open in [14]. Using SpLog, we answer this positively:

▶ **Theorem 28.** *Given a vsf-regex $\alpha$, an equivalent $\varphi \in$ SpLog can be computed in polynomial time.*

As a consequence, it is possible to extend the syntax of $\mathsf{SpLog}_{\mathsf{rx}}$, SpLog, and $\mathsf{EC}^{\mathsf{reg}}$ by defining constraints with vsf-regex instead of classical regular expressions, without affecting the complexity of evaluation or satisfiability (and core spanner representations can also use vsf-regex). Theorem 28 also shows that, given vsf-regex $\alpha_1, \ldots, \alpha_n$, one can decide in PSPACE whether $\bigcap \mathcal{L}(\alpha_i) = \emptyset$ (by converting each $\alpha_i$ into a formula $\varphi_i$, and deciding the satisfiability of $\bigwedge \varphi_i$). This is an interesting contrast to the full class of regex, where even the intersection emptiness problem for two languages is undecidable (cf. Carle and Narendran [3]).

## 4.4 A Normal Form for SpLog

Another advantage of using a logic is the existence of normal forms. In order to consider a short example of such an application, we introduce the following:

▶ **Definition 29.** A $\varphi \in$ SpLog is a *prenex conjunction* if $\varphi = \exists x_1, \ldots, x_k \colon (\bigwedge_{i=1}^m \eta_i \wedge \bigwedge_{j=1}^n C_j)$, with $k, n \geq 0$, $m \geq 1$, where the $\eta_i$ are word equations, and the $C_j$ are constraints. A SpLog-formula is in *DPC-normal form (DPCNF)* if it is a disjunction of prenex conjunctions.

▶ **Lemma 30.** *Given $\varphi \in$ SpLog, we can compute $\psi \in$ SpLog in DPCNF with $\varphi \equiv \psi$.*

Fagin et al. [12] also examined $\mathsf{CRPQ}^=$ and $\mathsf{UCRPQ}^=$ (conjunctive regular path queries with string equality, and unions of these). These are existential positive queries on graphs, but when restricted to marked paths, $[\![\mathsf{UCRPQ}^=]\!] = [\![\mathsf{RGX}^{\mathsf{core}}]\!]$ holds (cf. [12]). Using our methods, it is easy to show that there are polynomial time transformations between $\mathsf{CRPQ}^=$ and SpLog prenex conjunctions, and between $\mathsf{UCRPQ}^=$ and DPCNF-formulas. The author conjectures that the exponential blowup from the proof of Lemma 30 is necessary. This would immediately imply that there is an exponential blowup from $\mathsf{RGX}^{\mathsf{core}}$ to $\mathsf{UCRPQ}^=$.

We use DPCNF to illustrate some differences between SpLog and $\mathsf{EC}^{\mathsf{reg}}$: First, consider the following: Every $\mathsf{EC}^{\mathsf{reg}}$-formula $\varphi$ with $\mathsf{free}(\varphi) = \{x\}$ defines a language $\mathcal{L}(\varphi) := \{\sigma(x) \mid \sigma \models \varphi\}$ (in Section 5, we shall see that this has applications beyond the language theoretic point of view). For $\mathcal{C} \in \{\mathsf{EC}, \mathsf{EC}^{\mathsf{reg}}, \mathsf{SpLog}\}$, a language $L \subseteq \Sigma^*$ is a $\mathcal{C}$-language if there is a $\varphi \in \mathcal{C}$ with $\mathcal{L}(\varphi) = L$. We denote this by $L \in \mathcal{L}(\mathcal{C})$. For $L \subseteq \Sigma^*$ and $a \in \Sigma$, we define the *right quotient of $L$ by $a$* as $L/\mathtt{a} := \{w \mid wa \in L\}$. It is easily seen that the class of $\mathsf{EC}^{\mathsf{reg}}$-languages is closed under this operation, by using formulas like $\varphi_{/\mathtt{a}}(w) := \exists u \colon ((u = wa) \wedge \varphi(u))$. But as SpLog-variables can only contain subwords of the main variable, writing $u = wa$ is not possible in SpLog. The proof for the analogous is more involved and relies on Lemma 30.

▶ **Lemma 31.** *For every SpLog-language $L$ and every $a \in \Sigma$, $L/\mathtt{a}$ is a SpLog-language.*

This allows us to use Greibach's Theorem [17] to prove the following:

▶ **Proposition 32.** *The following conditions are equivalent:*
1. $\mathcal{L}(\mathsf{EC}^{\mathsf{reg}}) = \mathcal{L}(\mathsf{SpLog})$,
2. *Given* $\varphi \in \mathsf{EC}^{\mathsf{reg}}$*, it is decidable whether* $\mathcal{L}(\varphi) \in \mathcal{L}(\mathsf{SpLog})$*,*
3. $\mathcal{L}(\mathsf{SpLog})$ *is closed under the prefix operator.*

This characterization might serve as a starting point to answer whether $[\![\mathsf{EC}^{\mathsf{reg}}]\!] = [\![\mathsf{SpLog}]\!]$, an important question that is left open in the present paper (we define $[\![\mathcal{C}]\!] := \{[\![\varphi]\!] \mid \varphi \in \mathcal{C}\}$ for $\mathcal{C} \subseteq \mathsf{EC}^{\mathsf{reg}}$). The question appears to be surprisingly complicated; even when only considering word equations. We only discuss this briefly, as a deeper examination would require considerable additional notation. In contrast to EC and $\mathsf{EC}^{\mathsf{reg}}$, SpLog can only use variables that are subwords of the main variable. Hence, one might expect that it is easy to construct an EC-formula where other variables are necessary. But as it turns out, many word equations can be rewritten to reduce the number of variables. In particular, there is a notion of word equations where the solution set can be *parameterized* (i. e., expressed with a finite number of so-called parametric words – for more details, see e. g. Czeizler [6], Karhumäki and Saarela [22]). In all cases that were considered by the author, it was possible to use these parametrizations to construct SpLog-formulas. Similarly, the solution sets of non-parametrizable equations that the author examined, like $x\mathtt{ab}y = y\mathtt{ba}x$, are self-similar in a way that allows the construction of SpLog-formulas (cf. Czeizler [6], Ilie and Plandowski [19]). On the other hand, these constructions do not appear to generalize straightforwardly to an equivalence proof.

## 5 Using EC-Inexpressibility to Prove Non-Selectability

While Section 4 examined various aspects of expressing relations in SpLog, the present section examines how to prove that a relation cannot be selected. As we shall see, this can often be proved by using inexpressibility of appropriate languages. To this end, general tools for language inexpressibility (like a pumping lemma) would be very convenient. Up to now, the only (somewhat) general technique for core spanner inexpressibility was given in [14], where it was observed that on unary alphabets, core spanners can only define semi-linear (and, hence, regular) languages. Due to the limited applicability of this result, having further inexpressibility techniques appears to be desirable. As SpLog is a fragment of $\mathsf{EC}^{\mathsf{reg}}$, it is natural to ask whether this connection can be used to obtain inexpressibility results.

Karhumäki et al. [20] developed multiple inexpressibility techniques for EC. Sadly, EC-inexpressibility does not imply SpLog-inexpressibility; e. g., for $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, the language $\{\mathtt{a}, \mathtt{b}\}^*$ is not EC-expressible (cf. [20]), but obviously SpLog-expressible. On the other hand, while $\mathsf{EC}^{\mathsf{reg}}$-inexpressibility results would be useful, to the author's knowledge, the only result in this direction is that every $\mathsf{EC}^{\mathsf{reg}}$-language is an EDT0L-language (cf. Ciobanu et al. [4]). While this allows the use of the EDT0L-inexpressibility results (e. g. Ehrenfeucht and Rozenberg [11]), the large expressive power of EDT0L limits the usefulness of this approach.

As we shall see, developing a sufficient criterion for EC-expressible SpLog-languages allows us to use one of the techniques from [20] for SpLog. We begin with a definition: A language $L \subseteq \Sigma^*$ is *bounded* if there exist words $w_1, w_2, \ldots, w_n \in \Sigma^+$, $n \geq 1$, such that $L \subseteq w_1^* w_2^* \cdots w_n^*$. Combining a characterization of the class of bounded regular languages (Ginsburg and Spanier [16]) with the observations on EC from [20] yields the following:

▶ **Lemma 33.** *Every bounded regular language is an EC-language.*

▶ **Theorem 34.** *Every bounded SpLog-language is an EC-language.*

The intuition behind this is very simple: In bounded SpLog-languages, every constraint can be replaced with a bounded regular language (as this reasoning does not apply to $\mathsf{EC}^{\mathsf{reg}}$, the proof does not generalize). The EC-inexpressibility technique from [20] that we are going to use is based on the following definition by Karhumäki, Plandowski, and Rytter [21]:

▶ **Definition 35.** A word $w \in \Sigma^+$ is *imprimitive* if there exist a $u \in \Sigma^+$ and $n \geq 2$ with $w = u^n$. Otherwise, $w$ is *primitive*. For a given primitive word $Q$, the $\mathscr{F}_Q$-factorization of $w \in \Sigma^*$ is the factorization $w = w_0 \cdot Q^{x_1} \cdot w_1 \cdots Q^{x_k} \cdot w_k$ that satisfies the following conditions:

1. $Q^2 \not\sqsubseteq w_i$ for all $0 \leq i \leq k$,
2. $Q$ is a proper suffix of $w_0$, or $w_0 = \varepsilon$,
3. $Q$ is a proper prefix of $w_k$, or $w_k = \varepsilon$,
4. $Q$ is a proper prefix and a proper suffix of $w_i$ for all $0 < i < k$.

Furthermore, we define $T_Q(w) := \{x \mid Q^x \text{ occurs in the } \mathscr{F}_Q\text{-factorization of } w\}$, as well as $\exp_Q(w) := \max(T_Q(w) \cup \{0\})$.

For every primitive word $Q$, the $\mathscr{F}_Q$-factorization of every word $w$ (and, hence, $\exp_Q(w)$) is uniquely defined (cf. [20, 21]). We use this definition in the following pumping result:

▶ **Theorem 36** (Karhumäki et al. [20]). *For every EC-language $L$ and every primitive word $Q$, there exists a $k \geq 0$ such that, for each $w \in L$ with $\exp_Q(w) > k$, there is a $u \in L$ with $\exp_Q(u) \leq k$ which is obtained from $w$ by removing some occurrences of $Q$.*

We now consider a short example of this proof technique: As shown by Fagin et al. [12] (Theorem 4.21), $L_{\mathrm{el}} := \{\mathtt{a}^i\mathtt{b}^i \mid i \geq 0\}$ is not expressible with core spanners (note that $L_{\mathrm{el}}$ is also used in [20] as an example application of Theorem 36). The length of this proof is roughly one page. Contrast this to the following: Assume that $L_{\mathrm{el}}$ is a SpLog-language. Then $L_{\mathrm{el}}$ is an EC-language, due to Theorem 34. Choose the primitive word $Q := \mathtt{a}$. Then there exists a $k \geq 0$ that satisfies Theorem 36. Choose $w := \mathtt{a}^{k+2}\mathtt{b}^{k+2}$, and observe that $\exp_Q(w) = k+1 > k$ (due to the factorization $w = \varepsilon \cdot \mathtt{a}^{k+1} \cdot \mathtt{a}\mathtt{b}^{k+2}$). Hence there exists a $u = \mathtt{a}^{k+2-j}\mathtt{b}^{k+2}$, $j > 0$, with $u \in L_{\mathrm{el}}$. As $k+2-j < k+2$, this is a contradiction.

From the inexpressibility of $L_{\mathrm{el}}$, Fagin et al. then conclude that the equal length relation $\{(u,v) \mid |u| = |v|\}$ is not selectable with core spanners (Karhumäki et al. [20] and Ilie [18] use the same approach for EC: Show the non-selectability of a relation by proving that a suitable language is not expressible). Using Theorem 36 and 34 we observe:

▶ **Proposition 37.** *For $x, y \in \Sigma^*$, $x$ is a* scattered subword *of $y$ if there exist a $k \geq 1$, $x_1, \ldots, x_k, y_0, \ldots y_k \in \Sigma^*$ with $x = x_1 \cdots x_k$ and $y = y_0(x_1y_1) \cdots (x_ky_k)$. For every word $w \in \Sigma^*$, its* reversal *$w^R$ is the word that is obtained by reading $w$ from right to left. We define the following binary relations over $\Sigma^*$:*

$$R_{\mathrm{scatt}} := \{(u,v) \mid u \textit{ is a scattered subword of } v\}, \qquad R_{\mathrm{rev}} := \{(u,v) \mid v = u^R\},$$
$$R_{\mathrm{num}(a)} := \{(u,v) \mid |u|_a = |v|_a\} \textit{ for } a \in \Sigma, \qquad R_< := \{(u,v) \mid |u| < |v|\},$$
$$R_{\mathrm{permut}} := \{(u,v) \mid |u|_a = |v|_a \textit{ for all } a \in \Sigma\}.$$

*Each of $R_{\mathrm{scatt}}$, $R_{\mathrm{num}(a)}$, $R_{\mathrm{permut}}$, $R_{\mathrm{rev}}$, and $R_<$ is not SpLog-selectable.*

Sadly, being limited to bounded languages also limits the applicability of this approach. For example, Ilie [18] shows that the language of square-free words over a two letter alphabet (words that contain no subword $xx$ with $x \neq \varepsilon$) is not EC-expressible. Although one could expect that this language is not a SpLog-language, it is easily seen that every bounded subset of this language has to be finite, which means that this technique cannot be applied. Furthermore, the author conjectures that the relation $\{(x, x^n) \mid x \in \Sigma^*, n \geq 1\}$ is not SpLog-selectable, but there is no suitable bounded language that could be used to prove this.

## 6 Conclusions and Further Directions

As we have seen, SpLog has the same expressive power as the three classes of representations for core spanners that were introduced by Fagin et al. [12], and it is possible to convert between these models in polynomial time. As a result of this, core spanner representations can be converted to SpLog to decide satisfiability and hierarchicality, and SpLog provides a convenient way of defining core spanners, and in particular relations that are selectable by core spanners (see e. g. $\varphi_{\neq}$ in Example 27). Of course, whether one considers SpLog or one of the spanner representations more convenient mostly depends on personal preferences and the task at hand. Independent of one's opinion regarding the practical applications of SpLog, it can be used as a versatile tool for examining core spanners: For example, we used SpLog as intermediary to obtain polynomial time conversions between various subclasses of $\mathsf{VA}^{\mathsf{core}}$.

In addition to this, we defined a pumping lemma for core spanners by connecting SpLog to EC. A promising next step could be extending this to more general inexpressibility techniques that go beyond bounded SpLog languages. While the connection to word equations suggests that this line of research is difficult, one might also expect that at least some of the existing techniques for word equations can be used.

Another set of question where the comparatively simple syntax and semantics of SpLog might help is the relative succinctness of various models. For example, in order to examine the blowup from $\mathsf{VA}^{\mathsf{core}}$ to $\mathsf{RGX}^{\mathsf{core}}$, it suffices to examine the blowup from NFAs to $\mathsf{SpLog}_{\mathsf{rx}}$; and converting $\mathsf{RGX}^{\mathsf{core}}$ to $\mathsf{UCRPQ}^{=}$ has the same blowup as the transformation of SpLog-formulas to DPCNF. (Conjecture: All these blowups are exponential.)

Finally, the conversion of SpLog-formulas to spanner representations preserves many structural properties. Hence, when looking for subclasses of spanners that have certain properties (e. g., more efficient combined complexity of evaluation), the search can start with examining certain fragments of SpLog that correspond to interesting classes of spanners. One direction that seems to be promising as well as challenging is developing a notion of acyclic core spanners, which would need to account for the interplay of join and string equality (as seen in Corollary 23, every spanner representation can be rewritten into a representation that simulates $\bowtie$ with $\times$ and $\zeta^{=}$). This direction might be helped by first defining acyclicity for SpLog-formulas, which in turn could be inspired by the restrictions that are discussed in Reidenbach and Schmid [24].

A more fundamental question is whether $[\![\mathsf{EC}^{\mathsf{reg}}]\!] = [\![\mathsf{SpLog}]\!]$. In addition to our discussion in Section 4.4, a potential approach to this is examining whether every bounded $\mathsf{EC}^{\mathsf{reg}}$-language is an EC-language (as $\mathsf{EC}^{\mathsf{reg}}$ can use arbitrary variables, the reasoning from Theorem 34 does not carry over from SpLog to $\mathsf{EC}^{\mathsf{reg}}$).

Another aspect of SpLog that makes it interesting beyond its connection to core spanners is that it can be understood as the fragment of $\mathsf{EC}^{\mathsf{reg}}$ describes properties of words without using any additional space, as every variable and equation has to be a subword of the main variable (hence, the name "SpLog" can also be interpreted as "subword property logic"). One effect of this is that evaluation of SpLog has lower upper bounds that evaluation of $\mathsf{EC}^{\mathsf{reg}}$. While we have only defined SpLog with a single main variable, a natural generalization would be allowing multiple main variables (the definition generalizes naturally, and the upper bound for evaluation remains). A potential application of SpLog with two (or more) variables is describing relations for path labels in graph databases.

**Acknowledgements** The author thanks Wim Martens for helpful comments and discussions, and the anonymous reviewers for their insightful feedback.

## References


1. P. Barceló and P. Muñoz. Graph logics with rational relations: the role of word combinatorics. In *Proc. CSL-LICS 2014*, 2014.
2. J.-C. Birget. Intersection and union of regular languages and state complexity. *Inform. Process. Lett.*, 43(4):185–190, 1992.
3. B. Carle and P. Narendran. On extended regular expressions. In *Proc. LATA 2009*, 2009.
4. L. Ciobanu, V. Diekert, and M. Elder. Solution sets for equations over free groups are EDT0L languages. In *Proc. ICALP 2015*, 2015.
5. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.
6. Elena Czeizler. The non-parametrizability of the word equation $xyz{=}zvx$: A short proof. *Theor. Comput. Sci.*, 345(2-3):296–303, 2005.
7. V. Diekert. Makanin's Algorithm. In M. Lothaire, editor, *Algebraic Combinatorics on Words*, chapter 12. Cambridge University Press, 2002.
8. V. Diekert. More than 1700 years of word equations. In *Proc. CAI 2015*, 2015.
9. V. Diekert, A. Jeż, and W. Plandowski. Finding all solutions of equations in free groups and monoids with involution. In *Proc. CSR 2014*, 2014.
10. V. G. Durnev. Undecidability of the positive $\forall\exists^3$-theory of a free semigroup. *Sib. Math. J.*, 36(5):917–929, 1995.
11. A. Ehrenfeucht and G. Rozenberg. A pumping theorem for EDT0L languages. Technical report, Tech. Rep. CU-CS-047-74, University of Colorado, 1974.
12. R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Document spanners: A formal approach to information extraction. *J. ACM*, 62(2):12, 2015.
13. D. D. Freydenberger. Extended regular expressions: Succinctness and decidability. *Theory Comput. Sys.*, 53(2):159–193, 2013.
14. D. D. Freydenberger and M. Holldack. Document spanners: From expressive power to decision problems. In *Proc. ICDT 2016*, 2016.
15. M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman and Company, 1979.
16. S. Ginsburg and E. H. Spanier. Bounded regular sets. *Proc. AMS*, 17(5):1043–1049, 1966.
17. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
18. L. Ilie. Subwords and power-free words are not expressible by word equations. *Fundam. Inform.*, 38(1-2):109–118, 1999.
19. L. Ilie and W. Plandowski. Two-variable word equations. *ITA*, 34(6):467–501, 2000.
20. J. Karhumäki, F. Mignosi, and W. Plandowski. The expressibility of languages and relations by word equations. *J. ACM*, 47(3):483–505, 2000.
21. J. Karhumäki, W. Plandowski, and W. Rytter. Generalized factorizations of words and their algorithmic properties. *Theor. Comput. Sci.*, 218(1):123–133, 1999.
22. J. Karhumäki and A. Saarela. An analysis and a reproof of Hmelevskii's theorem. In *Proc. DLT 2008*, 2008.
23. G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.
24. D. Reidenbach and M. L. Schmid. Patterns with bounded treewidth. *Inform. Comput.*, 239:87–99, 2014.
25. M. L. Schmid. Characterising REGEX languages by regular languages equipped with factor-referencing. *Inform. Comput.*, 249:1–17, 2016.
26. R. Sedgewick and W. Wayne. *Algorithms*. Addison-Wesley, 2011.

# A Appendix – Proofs of the Results in This Paper

## A.1 Proof of Lemma 9

**Proof.** We first show the proof for vset-automata (the construction for vstk-automata only requires some minor modifications). Let $A = (Q, q_0, q_f, \delta)$ be a functional vset-automaton. We first ensure that every state of $A$ is reachable from $q_0$, and that $q_f$ is reachable from every state (this is possible in polynomial time, and all states that do not satisfy these criteria can be removed). In order to make some general observations on functional vset-automata, for every $q \in Q$, we define sets $O_q, C_q \subseteq \mathsf{SVars}(A)$ as follows:

$$O_q := \{x \in \mathsf{SVars}(A) \mid q \in \hat{\delta}(q_0, r) \text{ for some } r \in (\Sigma \cup \Gamma)^* \text{ with } |r|_{\vdash_x} = 1\},$$

$$C_q := \{x \in \mathsf{SVars}(A) \mid q \in \hat{\delta}(q_0, r) \text{ for some } r \in (\Sigma \cup \Gamma)^* \text{ with } |r|_{\dashv_x} = 1\}.$$

Intuitively, the sets represent which variables are opened (in the case of $O_q$) or closed (in the case of $C_q$) on a way from $q_0$ to $q$. As $A$ is functional, $\mathcal{R}(A) = \mathsf{Ref}(A)$ must hold. This allows us the following observation:

- If $x \in O_q$, then $|r|_{\vdash_x} = 1$ for all $r \in (\Sigma \cup \Gamma)^*$ with $q \in \hat{\delta}(q_0, r)$,
- if $x \in C_q$, then $|r|_{\dashv_x} = 1$ for all $r \in (\Sigma \cup \Gamma)^*$ with $q \in \hat{\delta}(q_0, r)$.

This is easily proven: Assume that for some $q \in Q$ and $x \in O_q$, there is an $r_1 \in (\Sigma \cup \Gamma)^*$ with $q \in \hat{\delta}(q_0, r_1)$ and $|r_1|_{\vdash_x} \neq 1$. As $q_f$ is reachable from $q$, there exists an $s \in (\Sigma \cup \Gamma)^*$ with $q_f \in \hat{\delta}(q, s)$, which implies $r_1 \cdot s \in \mathcal{R}(A)$. Due to $\mathsf{Ref}(A) = \mathcal{R}(A)$, $|r_1|_{\vdash_x} = 0$ and $|s|_{\vdash_x} = 1$ must hold. Furthermore, as $x \in O_q$, there is an $r_2 \in (\Sigma \cup \Gamma)^*$ with $q \in \hat{\delta}(q_0, r_2)$ and $|r_2|_{\vdash_x} = 1$. But then $r_2 \cdot s \in \mathcal{R}(A)$, and as $|r_2 \cdot s|_{\vdash_x} = 2$, this contradicts $\mathcal{R}(A) = \mathsf{Ref}(A)$. The argument for $C_q$ proceeds analogously.

Hence, in functional vset-automata, for each state it is uniquely defined which variables have been opened or closed, regardless which path was taken. This means that, when simulating $A$, we do not need to keep track of the variables, as this information can be derived from the states. Moreover, in order to compute $O_q$ or $C_q$ for any $q \in Q$, it suffices to pick any path from $q_0$ to $q$, and observe which variables have been opened or closed along this path. This means that all $O_q$ and $C_q$ can be computed in polynomial time. Moreover, we can assume with out loss of generality that $O_{q_f} = C_{q_f} = \mathsf{SVars}(A)$; as otherwise $\mathsf{Ref}(A) = \emptyset$ would hold.

Now, for every $w \in \Sigma^*$, a $(\mathsf{SVars}(A), w)$-tuple can correspond to multiple ref-words. For example, if $\mu(x) = \mu(y)$, the corresponding ref-word can contain $\vdash_x \vdash_y$ or $\vdash_y \vdash_x$. Given $\mu$, we can compute sequences $w_0, \ldots, w_n \in \Sigma^*$ and non-empty $M_1, \ldots, M_n \subseteq \Gamma$ for some $n \geq 0$ such that the following holds:

1. $w = w_0 w_1 \cdots w_n$,
2. $w_i \neq \varepsilon$ for $1 \leq i < n$,
3. $\bigcup_{i=1}^{n} M_i = \{\vdash_x, \dashv_x \mid x \in \mathsf{SVars}(A)\}$,
4. $\mu(x) = [o, c\rangle$ if and only if there exist $1 \leq i \leq j \leq n$ with $\vdash_x \in M_i$, $\dashv_x \in M_j$, $o = |w_0 \cdots w_{i-1}| + 1$, $c = |w_0 \cdots w_{j-1}| + 1$.

Intuitively, the sequences $w_0, \ldots, w_n \in \Sigma^*$ and $M_1, \ldots, M_n \subseteq \Gamma$ generalize all possible $r$ with $\mu^r = \mu$, in the following sense: If $r \in (\Sigma \cup \Gamma)^*$ with $\mu^r = \mu$, then for every $M_i$, the symbols in $M_i$ can be arranged into a word $r_i$ such that $r = w_0(r_1 w_1) \cdots (r_n w_n)$. As we require $w_i \neq \varepsilon$ and $S_i \neq \emptyset$, every pair $\mu$ and $w$ defines a unique pair of sequences $w_0, \ldots, w_n$ and $M_1, \ldots, M_n$.

In order to keep track of the opening and closing of variables during the simulation of $A$, we define

$$O_i := \bigcup_{j=1}^{i} \{\vdash_x \in M_j\}, \qquad C_i := \bigcup_{j=1}^{i} \{\dashv_x \in M_j\}$$

for all $1 \leq i \leq n$, as well as $O_0 := C_0 := \emptyset$. Intuitively, $O_i$ and $C_i$ shall represent the sets $O_q$ and $C_q$ for any $q$ that can be reached after processing $w_0(M_1 w_1)\cdots(M_i w_i)$.

We now simulate all possible $r$ with $\mu^r = \mu$ in a manner that is an extension of an on-the-fly computation of the powerset construction (for the simulation of NFAs with DFAs). To do so, we define $S_0 := \hat{\delta}(q_0, w_0)$. For $i$ from 1 to $n$, we iterate the following loop:

1. At the current point, our simulation of $A$ has processed the potential inputs up to $w_{i-1}$, i. e., we are in one of the states of $S_{i-1}$, and the status of the variables is found in $O_{i-1}$ and $C_{i-1}$. We now have to process any ordering of $M_i$, and we know that the intended resulting sets of open and closed variables can be found in $O_i$ and $C_i$. To this end, we define $S_{new}$ as the set of all $p \in Q$ such that $O_p = O_i$, $C_p = C_i$, and there exists a $q \in S_{i-1}$ such that $p$ can be reached from $q$ by taking only transitions with labels from $M_i \cup \{\varepsilon\}$. In other words, $S_{new}$ contains exactly the states that can be reached by processing all potential inputs up to $M_i$, and it can be computed with a straightforward reachability analysis. Recall that this reachability analysis does not need to take into account whether the path contains each element of $M_i$ exactly once, as this is handled by the sets $C_p$.
2. Define $S_i := \bigcup_{p \in S_{new}} \hat{\delta}(p, w_i)$. Then $S_i$ contains the states that can be reached by processing the potential inputs up to $w_i$, which means that we can proceed to the next iteration of the loop.

After computing $S_n$, we only need to check whether $q_f \in S_n$. This holds if and only if there is an $r \in \mathcal{R}(A)$ with $\mu^r = \mu$. Hence, we can decide $\mu \in [\![A]\!](w)$ in polynomial time, which proves the claim.

**vstk-automata** For vstk-automata, first observe that the states of functional vstk-automa store less information than vset-automata, which does not allow us to define $C_q$ in the same way. More specifically, for vstk-automata we do not know exactly which variables have been closed on the path to $q$, only how many. As a consequence, we define $N_q := |r|_{\dashv}$, where we can choose any $r \in (\Sigma \cup \Gamma)^*$ with $q \in \hat{\delta}(q_0, r)$. Similar to vset-automata, all such $r$ have to yield the same $N_q$.

Furthermore, instead of storing different $\dashv_x$ in the sets $M_i$, and using these compute $C_i$ for every step $i$, we shall directly compute sets $N_i$ that determine how many Next, note that we might have to split up some of the $M_i$: If there are variables $x, y$ with $\mu(x) = [k, l\rangle$ and $\mu(y) = [k, m\rangle$ and $l < m$, we know that $y$ is closed after $x$. Hence, $y$ must be opened before $x$ is opened, which means that $x$ and $y$ cannot remain in the same set $M_i$. Still, this can be derived directly from $\mu$, and we can include this in our construction by allowing $w_i = \varepsilon$ in those cases.

We then proceed analogously to the vset-construction by processing the $w_i$ as in the simulation of an NFA, and the sets $M_i$ with a reachability analysis, where the sets $C_i$ and $N_i$ determine which states are viable destinations. ◀

## A.2 Proof of Proposition 10

**Proof.** We are going to prove the following claim: Given $A \in \mathsf{VA}$ with $n$ states, $m$ edges, and $k$ variables, we can decide in time $O(km + n)$ whether $A$ is functional. We first discuss the algorithm for vset-automata; and then examine how it can be adapted to vstk-automata.

**Algorithm for vset-automata:** We use Algorithm 1. To understand this algorithm, recall that a vset-automaton $A$ is functional if and only if $\mathsf{Ref}(A) = \mathcal{R}(A)$, or, in other words, every word in $\mathcal{R}(A)$ is valid. Assuming that $A$ contains no redundant states (i. e., states that cannot be reached from $q_0$, or from which $q_f$ cannot be reached), this requires that for each state $q$ and all ref-words $r, \hat{r}$, $q \in \delta(q_0, r) \cap \delta(q_0, \hat{r})$ if and only if $|r|_g = |\hat{r}|_g$ for all $g \in \Gamma$ (otherwise, we could complete at least one of the two ref-words to a ref-word that is accepted by $A$, but is invalid).

Hence, if $A$ is functional, we can define the following two sets:

$$\mathsf{opened}(q) := \{x \mid \vdash_x |r|_{\vdash_x} = 1 \text{ for all } r \text{ with } q \in \delta(q_0, r)\},$$
$$\mathsf{closed}(q) := \{x \mid \vdash_x |r|_{\dashv_x} = 1 \text{ for all } r \text{ with } q \in \delta(q_0, r)\}.$$

Intuitively spoken, the two sets describe which variables have been opened and closed, respectively (note that the choice of "opened" as opposed to "open" is intentional: A variable remains in opened, even after it has been closed, as our goal is that for $q_f$, both sets are equal to $\mathsf{SVars}(A)$). The main idea behind Algorithm 1 is that we compute these sets successively for each state of $A$ (by following a depth-first-search). For each edge, we check whether the sets of the next state $p$ are consistent with the sets of the current state $q$: If the edge has a terminal label, the sets have to be identical. If the set has a label $\vdash_x$, then $x$ cannot have been opened in $q$, but must be open in $p$. Likewise, if the edge has a label $\dashv_x$, $x$ must be open in $q$, but closed in $p$. This ensures that every variable is opened at most once, and only closed after it has been opened. Whenever the algorithm finds an edge that connects two states where these sets are not consistent, we know that $A$ is not functional and can abort. After all edges have been checked, the algorithm ensures that every variable has been closed (as a variable can only be closed after it has been opened, this also guarantees that each variable has been opened and closed exactly once).

Regarding the complexity, let $n$ denote the number of states of the input vset-automaton $A$, let $m$ denote the number of its edges, and let $k := |\mathsf{SVars}(A)|$. The first two steps, eliminating the redundant states, can be performed in time $O(m + n)$ with a standard reachability analysis by breadth-first search (see Chapter 22 of Cormen et al. [5]). The complexity analysis for the rest of Algorithm 1 proceeds similarly: The while loop touches each of the $m$ edges at most once, hence there are at most $O(m)$ executions of the while loop (if $m < n$, then the first two steps have eliminated unreachable nodes; hence we can assume $m \geq n$ at this part of the algorithm). The operations on the sets (copying, adding, and equality test) take at most time $O(k)$, hence the whole algorithm runs in time $O(km + n)$.

**Algorithm for vstk-automata:** Algorithm 1 can be adapted to cover vstk-automata. The crucial difference is that, instead of maintaining a set of closed variables $\mathsf{closed}(q)$ for each $q \in Q$, it suffices to store the number of closed variables for $q$ in variable $\mathsf{closed}(q)$. Accordingly, we define $\mathsf{closed}(q_0) := 0$ in line 3. Most changes are applied to the for-loop in lines 24 to 31: First, instead of looping over all $p \in \delta(q, \dashv_x)$, we loop over all $p \in \delta(q, \dashv)$. Next, the rejection criterion in line 25 is changed to $\mathsf{closed}(q) \geq |\mathsf{opened}(q)$, as a functional automaton may only close a variable if sufficiently many have been opened. Accordingly, the condition $\mathsf{closed}(p) \neq \mathsf{closed}(q) \cup \{x\}$ in line 27 is changed to $\mathsf{closed}(p) \neq \mathsf{closed}(q) + 1$, as $p$ must have exactly one more closed variable than $q$. Due to the same reasoning, we change the assignment in line 31 to $\mathsf{closed}(p) \neq \mathsf{closed}(q) + 1$.

All that remains is to adapt the final check in line 32 to $\mathsf{closed}(q_f) = |\mathsf{SVars}(A)|$, as the accepting state must ensure that the correct number of variables has been closed. These

**Algorithm 1:** Functionality test for vset-automata

```
    Input: vset-automaton A = (Q, δ, q_0, q_f)
    Output: True if A is functional, False if A is not functional
 1  eliminate all states that are not reachable from q_0;
 2  eliminate all states from which q_f cannot be reached;
 3  opened(q_0) := ∅; closed(q_0) := ∅;
 4  forall q ∈ Q − {q_0} do
 5      opened(q) := undefined; closed(q) := undefined;
 6  visited := {q_0}; process := {q_0};
 7  while process ≠ ∅ do
 8      choose q ∈ process; remove q from process;
 9      foreach p ∈ δ(q, a) for some a ∈ Σ do
10          if p ∈ visited then
11              if opened(p) ≠ opened(q) or closed(p) ≠ closed(q) then
12                  return False
13          else
14              add p to visited and to process;
15              opened(p) := opened(q); closed(p) := closed(q);
16      foreach p ∈ δ(q, ⊢_x) for some x ∈ SVars(A) do
17          if x ∈ opened(q) then return False;
18          if p ∈ visited then
19              if opened(p) ≠ opened(q) ∪ {x} or closed(p) ≠ closed(q) then
20                  return False
21          else
22              add p to visited and to process;
23              opened(p) := opened(q) ∪ {x}; closed(p) := closed(q);
24      foreach p ∈ δ(q, ⊣_x) for some x ∈ SVars(A) do
25          if x ∉ opened(q) or x ∈ closed(q) then return False;
26          if p ∈ visited then
27              if opened(p) ≠ opened(q) or closed(p) ≠ closed(q) ∪ {x} then
28                  return False
29          else
30              add p to visited and to process;
31              opened(p) := opened(q); closed(p) := closed(q) ∪ {x};
32  if closed(q_f) = SVars(A) then
33      return True
34  else
35      return False
```

modifications do not change the running time of the algorithm; hence, the complexity remains unchanged.

◀

## A.3 Proof of Lemma 11

**Proof.** Membership in NP is obvious: To show that $\mathsf{Ref}(A,\varepsilon) \neq \emptyset$, it suffices to guess an path of length $2|\mathsf{SVars}\,(A)|$ through $A$, and to verify that this path corresponds to a word from $\mathsf{Ref}(A,\varepsilon)$.

We show NP-hardness by a basic reduction from the Hamiltonian path problem, which is defined as follows: Given a directed graph $G = (V, E)$, does $G$ contain a Hamiltonian path? (A Hamiltonian path is a sequence $(i_1, \ldots, i_n)$ with $n = |V|$, $i_1, \ldots, i_n \in V$, and $(i_j, i_{j+1}) \in E$ for all $1 \leq j < k$, such that for each $v \in V$, there is exactly one $j$ with $i_j = v$.)

We begin with the construction for vset-automata. Given a directed graph $G = (V, E)$, we construct an $A \in \mathsf{VA}_{\mathsf{set}}$ such that $[\![A]\!](\varepsilon) \neq \emptyset$ if and only if $G$ contains a Hamiltonian path. Assume that $V = \{1, \ldots, k\}$ for some $k \geq 1$. We shall define $A$ with $\mathsf{SVars}\,(A) = \{x_1, \ldots, x_k\}$. Let $A := (Q, \delta, q_0, F)$, with $Q := \{q_0, q_F\} \cup \{q_i \mid 1 \leq i \leq k\}$, $F := \{q_F\}$, and define $\delta$ as follows:

$$
\begin{aligned}
\delta(q_0, \vdash_{x_j}) &:= \{q_j\} \text{ for all } 0 \leq i \leq k,\\
\delta(q_i, \vdash_{x_j}) &:= \{q_j\} \text{ for all } (i,j) \in E,\\
\delta(q_i, \dashv_{x_j}) &:= \{q_F\} \text{ for all } 1 \leq i \leq k,\ 1 \leq j \leq k,\\
\delta(q_F, \dashv_{x_j}) &:= \{q_F\} \text{ for all } 1 \leq j \leq k.
\end{aligned}
$$

The intuition behind $A$ is as follows: Every state $q_j$ corresponds to the node $j$ of $G$, and it can only be entered by reading $\vdash_{x_j}$. Hence, for every edge $(i,j) \in E$, $A$ contains a transition from $q_i$ to $q_j$ that is labeled with $\vdash_{x_i}$. Finally, by reading any $\dashv_{x_j}$, $A$ can change to the accepting state, where it closes all remaining variables.

Hence, $\mathcal{R}(A)$ is the language of words $r = \vdash_{x_{i_1}} \vdash_{x_{i_2}} \cdots \vdash_{x_{i_n}} \cdot\, c$, where $c \in \{\dashv_{x_j} \mid 1 \leq j \leq k\}^*$, $n \geq 1$, $i_1, \ldots, i_n \in V$, and $(i_j, i_{j+1}) \in E$ for all $1 \leq j < k$. This means that we can interpret $r$ as a path $(i_1, \ldots, i_n)$ in $G$; and for every path, we can construct a corresponding word.

Moreover, if $r \in \mathsf{Ref}(A)$, then each node $\vdash_{x_i}$ has to occur exactly once in $r$, which means that the path $(i_1, \ldots, i_n)$ is a Hamiltonian path. Likewise, every Hamiltonian path can be used to construct a word from $\mathsf{Ref}(A)$.

As no transition of $A$ is labeled with a letter from $\Sigma$, $\mathsf{Ref}(A) = \mathsf{Ref}(A,\varepsilon)$. Hence, $\mathsf{Ref}(A,\varepsilon) \neq \emptyset$ if and only if $G$ contains a Hamiltonian path. As the Hamiltonian path problem is NP-complete (cf. Garey and Johnson [15]), this means that deciding emptiness of $\mathsf{Ref}(A,\varepsilon)$ is NP-hard. For vstk-automata, we can use the same construction and replace each $\dashv_{x_i}$ with $\dashv$. ◀

## A.4 Lemma 38 with Proof

Given a finite $V \subset \Xi$, we extend the notion of valid ref-words to $(\Sigma \cup \Xi)^*$ by calling a ref-word $r$ valid if it contains each of $\vdash_x$ and $\dashv_x$ exactly once and in the right order for all $x \in V$. We use the following result in Sections A.5 and A.6.

▶ **Lemma 38.** *For a finite $V \subset \Xi$, consider any valid $r \in (\Sigma \cup \Gamma_V)^*$ that contains no subword from $\Gamma_V^2$. Then for every valid $\hat{r} \in (\Sigma \cup \Gamma_V)^*$ with $\mathsf{clr}(r) = \mathsf{clr}(\hat{r})$, $\mu^{\hat{r}} = \mu^r$ implies $\hat{r} = r$.*

**Proof.** Choose any finite $V \subset \Xi$ and any valid $r \in (\Sigma \cup \Gamma_V)^*$ such that no $r$ contains no subword from $\Gamma_V^2$. Now consider any valid $\hat{r} \in (\Sigma \cup \Gamma_V)^*$ with $\mathsf{clr}(r) = \mathsf{clr}(\hat{r})$ and $\mu^{\hat{r}} = \mu^r$.

We first show that $\hat{r}$ also cannot contain a subword from $\Gamma_V^2$. In order to do so, first make an observation for all $x_1, x_2 \in V$ with $x_1 \neq x_2$. Let $[i_l, j_l\rangle := \mu^r(x)$ for $l \in \{1, 2\}$. Then the following holds:

1. $i_1 + 1 \neq j_1$ and $i_2 + 1 \neq j_2$,
2. $i_1 \neq i_2$ and $j_1 \neq j_2$,
3. $i_1 + 1 \neq j_2$ and $i_2 + 1 \neq j_1$.

We prove the first of these observations: Assume that there is an $x$ with $\mu(x) = [i, i+1\rangle$. Then $r = r_1 {\vdash_x} r_2 {\dashv_x} r_3$, and $r_2 \in \Gamma_V^*$, which means that $r$ contains a subword from $\Gamma_V^2$ : If $r_2 = \varepsilon$, then $r$ contains ${\vdash_x}{\dashv_x}$; and if $r_2 \neq \varepsilon$, then ${\vdash_x}$ and the leftmost letter (or ${\dashv_x}$ and the rightmost letter) of $r$ form a subword from $\Gamma_V^2$. Either way, we arrive at a contradiction. The remaining observations follow from analogous reasoning.

In order to prove that $\hat{r}$ contains no subword from $\Gamma_V^2$, assume that $g_1 \cdot g_2 \sqsubseteq \hat{r}$ with $g_1, g_2 \in \Gamma_V$. Then there exist $x_1, x_2 \in V$ with $g_l \in \{{\vdash_{x_l}}, {\dashv_{x_l}}\}$ for $l \in \{1, 2\}$. Let $[i_l, j_l\rangle := \mu^{\hat{r}}(x_l) = \mu^r(x_l)$. If $g_1 = {\vdash_{x_1}}$ and $g_2 = {\vdash_{x_2}}$, then $i_1 = i_2$, which contradicts the observations above. If $g_1 = {\vdash_{x_1}}$ and $g_2 = {\dashv_{x_2}}$, then $i_1 = j_2 - 1$, which also contradicts the observations. The two other possible choices for $g_1$ and $g_2$ lead to analogous contradictions. Hence, we conclude that $\hat{r}$ contains no subword from $\Gamma_V^2$.

Now assume that $\hat{r} \neq r$. Let $w := \mathsf{clr}(r)$ (recall that $\mathsf{clr}(r) = \mathsf{clr}(\hat{r})$ holds by our choice of $\hat{r}$). We now factorize both ref-words according to the leftmost distinct letter, i. e., let $r = p \cdot e \cdot s$ and $\hat{r} = p \cdot \hat{e} \cdot \hat{s}$, with $e, \hat{e} \in (\Sigma \cup \Gamma)$ and $e \neq \hat{e}$. Now, we distinguish three cases:

Firstly, assume $e \in \Sigma$ and $\hat{e} \in \Sigma$. Then the requirement $e \neq \hat{e}$ leads to $\mathsf{clr}(r) \neq \mathsf{clr}(\hat{r})$, which contradicts $\mathsf{clr}(r) = \mathsf{clr}(\hat{r}) = w$.

Secondly, assume $e \in \Gamma_V$ and $\hat{e} \in \Sigma$. Then there is an $x \in \mathsf{SVars}(A)$ with $e \in \{{\vdash_x}, {\dashv_x}\}$, and $e$ occurs somewhere in $\hat{s}$ (as $e$ cannot occur in $p$, and $e \neq \hat{e}$). But then we arrive at the contradiction $\mu^r(x) \neq \mu^{\hat{r}}(x)$. Analogously, $e \in \Sigma$ and $\hat{e} \in \Gamma_V$ leads to a contradiction.

Finally, assume $e \in \Gamma_V$ und $\hat{e} \in \Gamma_V$. Again, there is an $x \in \mathsf{SVars}(A)$ with $e \in \{{\vdash_x}, {\dashv_x}\}$, and $e$ must occur somewhere in $\hat{s}$. But as $\hat{r}$ contains no subword from $\Gamma_V^2$, the leftmost letter of $\hat{s}$ must belong to $\Sigma$. Hence, $\mu^r(x) \neq \mu^{\hat{r}}(x)$ must hold, which is a contradiction.

As we have exhausted all possibilities, we conclude that $\hat{r} = r$ must hold. ◀

The proof of Lemma 38 also applies to ref-words that use $\dashv$ instead of specific ${\dashv_x}$ (as they are used for vstk-automata). In particular, as every ref-word with $\dashv$ can be rewritten into an equivalent ref-word over $\Gamma_V$, the analogous result for ref-words with $\dashv$ follows immediately.

## A.5 Proof of Proposition 12

The proof of this proposition and of Proposition 13 both use the following result by Birget [2]:

▶ **Lemma 39** (Birget [2]). *Let $L$ be a regular language. Assume there exist pairs of words $(u_1, v_1), \ldots, (u_n, v_n)$ such that*

1. *$u_i v_i \in L$ for $1 \leq i \leq n$, and*
2. *$u_i v_j \notin L$ or $u_j v_i \notin L$ for all $1 \leq i < j \leq n$.*

*Then any NFA accepting $L$ must have at least $n$ states.*

In addition to this, we use Lemma 38, which is given in Section A.4. We are now ready to proceed to the proof of Proposition 12:

**Proof.** This proof is organized as follows: We first discuss vset-automata, then vstk-automata. For each of these, we first discuss upper and then lower bounds.

**Upper bound for vset-automata:** Consider a vset-automaton $A = (Q, \delta, q_0, q_f)$ with $k \geq 1$ variables. Our goal is to construct a functional vset-automaton $A_F$ with $3^k|Q|$ states and $[\![A_F]\!] = [\![A]\!]$. The main idea is to intersect $A$ with a functional vset-automaton that defines the universal spanner for $\mathsf{SVars}(A)$. Formally, we associate each state of $A_F$ with a function $s\colon \mathsf{SVars}(A) \to \{\mathtt{w}, \mathtt{o}, \mathtt{c}\}$, where $s(x)$ represents the following:

- $\mathtt{w}$ stands for "waiting", meaning that $\vdash_x$ has not been read,
- $\mathtt{o}$ stands for "open", meaning that $\vdash_x$ has been read, but $\dashv_x$ has not been read,
- $\mathtt{c}$ stands for "closed", meaning that $\vdash_x$ and $\dashv_x$ have been read.

Let $S$ be the set of all such functions. Observe that $|S| = 3^k$. We now define $A_F := (Q_F, \delta_F, q_{0,F}, q_{f,F})$ in the following way:

- $Q_F := Q \times S$,
- $q_{0,F} := (q_0, s_0)$, where $s_0$ is defined by $s_0(x) = \mathtt{w}$ for all $x \in \mathsf{SVars}(A)$,
- $q_{f,f} := (q_f, s_F)$, where $s_F$ is defined by $s_F(x) = \mathtt{c}$ for all $x \in \mathsf{SVars}(A)$,
- $\delta_F$ is defined as follows:
- $\delta_F((q, s), a) := \{(p, s) \mid p \in \delta(q, a)\}$ for all $a \in \Sigma$ and all $(q, s) \in Q_F$,
- for all $(q, s) \in Q_F$ and all $x \in \mathsf{SVars}(A)$, let

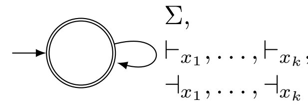
$$\delta_F((q, s), \vdash_x) := \begin{cases} \emptyset & \text{if } s(x) \neq \mathtt{w}, \\ \{(p, t_o) \mid p \in \delta(q, \vdash_x)\} & \text{if } s(x) = \mathtt{w}, \end{cases}$$

$$\delta_F((q, s), \dashv_x) := \begin{cases} \emptyset & \text{if } s(x) \neq \mathtt{o}, \\ \{(p, t_c) \mid p \in \delta(q, \dashv_x)\} & \text{if } s(x) = \mathtt{o}, \end{cases}$$

  where $t_o$ is defined by $t(x) := \mathtt{o}$ and $t_o(y) := s(y)$ for all $y \neq x$, and $t_c$ is defined by $t_c(x) := \mathtt{c}$ and $t_c(y) := s(y)$ for all $y \neq x$.

In order to see that $A_K$ is correct and functional, note that $A_F$ simulates $A$, while the definition of $\delta_F$ ensures that each variable $x$ can only be opened if $s(x) = \mathtt{w}$, and only be closed if $s(x) = \mathtt{o}$. As all variables start with $s(x) = \mathtt{w}$, and as $A_F$ only accepts if all variables hae $s(x) = \mathtt{c}$, this ensures that every variable is opened and closed exactly once. Finally, as $A_F$ has exactly $3^k|Q|$ states, this proves the upper bound for vset-automata.

**Lower bound for vstk-automata:** Let $k \geq 1$ and $X_k := \{x_1, \ldots, x_k\} \subset \Xi$. We define the following vset-automaton $A_k$:




In the terminology of [12], $A_k$ defines the universal spanner over $X_k$. Let $\mathtt{a} \in \Sigma$. Recall the set $S$ of all functions $s\colon X_k \to \{\mathtt{w}, \mathtt{o}, \mathtt{c}\}$, which we already used for the upper bound above. For every $s \in S$, we define ref-words $u_s := u_1^s \cdots u_k^s$ and $v_s := v_1^s \cdots v_k^s$, where the words $u_i^s$ and $v_i^s$ are defined as follows for every $i$, $1 \leq i \leq k$:

$$u_i^s := \begin{cases} \varepsilon & \text{if } s(x_i) = \mathtt{w}, \\ \vdash_{x_i}\mathtt{a} & \text{if } s(x_i) = \mathtt{o}, \\ \vdash_{x_i}\mathtt{a}\dashv_{x_i}\mathtt{a} & \text{if } s(x_i) = \mathtt{c} \end{cases} \qquad v_i^s := \begin{cases} \vdash_{x_i}\mathtt{a}\dashv_{x_i}\mathtt{a} & \text{if } s(x_i) = \mathtt{w}, \\ \dashv_{x_i}\mathtt{a} & \text{if } s(x_i) = \mathtt{o}, \\ \varepsilon & \text{if } s(x_i) = \mathtt{c} \end{cases}$$

Now observe that for each $s \in S$, $u_s \cdot v_s \in \mathsf{Ref}(A_k)$. Furthermore, $u_s \cdot v_s$ does not contain any subword from $\Gamma^2$. Hence, according to Lemma 38, for every vset-automaton $A$ with $[\![A]\!] = [\![A_k]\!]$, $u_s v_s \in \mathsf{Ref}(A)$ must hold.

Let $A \in \mathsf{VA_{set}}$ be functional with $[\![A]\!] = [\![A_k]\!]$. As $A$ is functional, $\mathsf{Ref}(A) = \mathcal{R}(A)$, which implies $u_s \cdot v_s \in \mathcal{R}(A)$ for all $s \in S$. Furthermore, for all $s, t \in S$ with $s \neq t$, $u_s \cdot v_t \notin \mathcal{R}(A)$ must hold, as $u_s \cdot v_t$ is not a valid ref-word. In order to see this, consider an $x_i$ with $s(x_i) \neq t(x_i)$. As each $\vdash_{x_i}$ and $\dashv_{x_i}$ occurs only in $u_i^s$ and $v_i^t$, $u_s \cdot v_t$ cannot contain both of $\vdash_{x_i}$ and $\dashv_{x_i}$ exactly once.

This allows us to use Lemma 39: For each $s \in S$, $u_s \cdot v_s \in \mathcal{R}(A)$; but for each $t \in S$ with $t \neq s$, $u_s \cdot v_t \notin \mathcal{R}(A)$. Hence, $A$ has at least $|S| = 3^k$ states. As $A$ was chosen freely among functional vset-automata, this proves the claimed lower bound.

**Upper bound for vstk-automata:** Consider a vstk-automaton $A = (Q, \delta, q_0, q_f)$ with $k \geq 1$ variables. Our goal is to construct a functional vstk-automaton $A_F$ with $(k+2)2^{k-1}|Q|$ states and $[\![A_F]\!] = [\![A]\!]$. On a conceptual level, the construction is very similar to the vset-automata construction above. The only difference is what information on the variables has to be maintained in the storage. For vstk-automata, we store which variables have been opened (in order to ensure that every variable is opened exactly once), and how many variables have been closed (in order to ensure that every variable is closed at least once, and to prevent processing $\dashv$ when no variables can be closed). We now define $A_F := (Q_F, \delta_F, q_{0,F}, q_{f,F})$ in the following way:

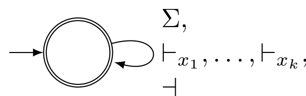
- $Q_F := \{(q, O, i) \mid q \in Q, O \subseteq \mathsf{SVars}(A), 0 \leq i \leq |O|\}$,
- $q_{0,F} := (q_0, \emptyset, 0)$,
- $q_{f,F} := (q_f, \mathsf{SVars}(A), k)$,
- $\delta_F((q, O, i), a) := \{(p, O, i) \mid p \in \delta(q, a)\}$ for all $a \in \Sigma$ and all $(q, O, i) \in Q_F$,
- for all $(q, O, i) \in Q_F$ and all $x \in \mathsf{SVars}(A)$, let

$$\delta_F((q, O, i, \vdash_x) := \begin{cases} \emptyset & \text{if } x \in O, \\ \{(p, O \cup \{x\}, i) \mid p \in \delta(q, \vdash_x)\} & \text{if } x \notin O, \end{cases}$$

$$\delta_F((q, O, i), \dashv) := \begin{cases} \emptyset & \text{if } i \geq |O|, \\ \{(p, O, i+1) \mid p \in \delta(q, \dashv)\} & \text{if } i < |O| \end{cases}$$

It is now easy to see that $A_F$ simulates $A$. In addition to this, the definition of $\delta_F$ ensures that variables are only opened if they have not been opened before (as $x \notin O$ is required in order to process $\vdash_x$), and that variables can only be closed if there are sufficiently many open variables (as $i < |O|$ is required to process $\dashv$). Furthermore, $A_F$ accepts only if every variable has been opened, and if $k$ variables have been closed. Hence, $A_F$ is functional and equivalent to $A$. All that remains for this upper bound is to prove that $|Q_F| = (k+2)2^{k-1}|Q|$. First, note that in the definition of $Q_F$, each state of $Q$ paired with an element of the set $M := \{(O, i) \mid O \subseteq \mathsf{SVars}(A), i \leq |O|\}$. We observe that $|M| = \sum_{j=0}^{k} \binom{k}{j}(j+1)$, as there are $\binom{k}{j}$ possible sets $O$ with $|O| = j$; and for each such set. By simplifying this formula, we obtain $|M| = (k+2)2^{k-1}$. (The author used Maxima and Wolfram Alpha to find this closed form.) As $Q_F = |M||Q|$, this concludes the proof of the upper bound.

**Lower bound for vstk-automata:** Again, this proof is similar to the vstk-case. Let $k \geq 1$ and $X_k := \{x_1, \ldots, x_k\} \subset \Xi$. We define the following vset-automaton $A_k$:



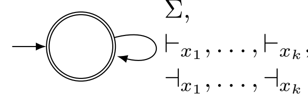
In the terminology of [12], $A_k$ defines the universal hierarchical spanner over $X_k$. Let $\mathsf{a} \in \Sigma$. Again, we want to define a sequence of pairs of ref-words that allows us to use Lemma 39. Recall the set $M := \{(O,i) \mid O \subseteq X_k, i \leq |O|\}$ that we already used in the proof for the upper bound. For each $(O,i) \in M$, we define ref-words $u_{O,i} := u_1^O \cdots u_k^O (\dashv \mathsf{a})^i$ and $v_{O,i} := u_1^O \cdots v_k^O (\dashv \mathsf{a})^{k-i}$ by

$$u_j^O := \begin{cases} \vdash_{x_j} \mathsf{a} & \text{if } x_j \in O, \\ \varepsilon & \text{if } x_j \notin O \end{cases} \qquad v_j^O := \begin{cases} \varepsilon & \text{if } x_j \in O, \\ \vdash_{x_j} \mathsf{a} & \text{if } x_j \notin O \end{cases}$$

for all $j$ with $1 \leq j \leq k$. First, observe that $u_{O,i} \cdot v_{O,i} \in \mathsf{Ref}(A_k)$ holds for all $(O,i) \in M$. Assume that $A$ is a functional vstk-automaton with $[\![A]\!] = [\![A_k]\!]$. As Lemma 38 applies (see the remark its proof), we know that $u_{O,i} \cdot v_{O,i} \in \mathcal{R}(A)$ holds for all $(O,i) \in M$. Next, consider $(O,i),(O',i') \in M$ with $(O,i) \neq (O',i')$ and let $r := u_{O,i} \cdot v_{O',i'}$. Then $r \notin \mathcal{R}(A)$ must hold: If $i \neq i'$, then $r$ contains too many or too few occurrences of $\dashv$. If $O \neq O'$, then a variable $x$ is opened more than once (if $x \in O$ and $x \notin O'$) or less than once (if $x \notin O$ and $x \in O'$). In each of these cases, $r$ is not valid, which contradicts our assumption that $A$ is functional. Hence, we can apply Lemma 39, and conclude that $A$ has at least $|M|$ states. As we established in the proof of the upper bound, $|M| = (k+2)2^{k-1}$. ◀

## A.6 Proof of Proposition 13

**Proof.** Let $\mathsf{a} \in \Sigma$, $k \geq 1$, and $X_k := \{x_1, \ldots, x_k\} \subset \Xi$. We use the same vstk-automaton $A_k$ as in the proof of the lower bound for vstk-automata in Proposition 12:




We now particularly focus on the following subset of $\mathsf{Ref}(A_k)$:

$$R_k := \left\{ (\vdash_{x_{p(1)}} \mathsf{a}) \cdots (\vdash_{x_{p(k)}} \mathsf{a})(\dashv \mathsf{a})^k \mid p \in \mathsf{Perm}(k) \right\},$$

where $\mathsf{Perm}(k)$ is the set of all permutations of $\{1, \ldots, k\}$. Translating these ref-words to ref-words that use explicit closing commands, we obtain the language

$$R'_k := \left\{ (\vdash_{x_{p(1)}} \mathsf{a}) \cdots (\vdash_{x_{p(k)}} \mathsf{a})(\dashv_{x_{p(k)}} \mathsf{a}) \cdots (\dashv_{x_{p(1)}} \mathsf{a}) \mid p \in \mathsf{Perm}(k) \right\}.$$

As $R'_k$ makes the closing of variables explicit, we can state that for every $r \in R_k$, there is an $r' \in R'_k$ with $\mu^r = \mu^{r'}$, and vice versa. Hence, for every vset-automaton $A$ with $[\![A]\!] = [\![A_k]\!]$, Lemma 38 implies that $R'_k \subseteq \mathcal{R}(A)$. For every permutation $p \in \mathsf{Perm}(k)$, we now define

$$u_p := (\vdash_{x_{p(1)}} \mathsf{a}) \cdots (\vdash_{x_{p(k)}} \mathsf{a}),$$
$$v_p := (\dashv_{x_{p(k)}} \mathsf{a}) \cdots (\dashv_{x_{p(1)}} \mathsf{a}).$$

For every $p \in \mathsf{Perm}(k)$, $u_p v_p \in R'_k$ holds, which implies $u_p v_p \in \mathcal{R}(A)$. Now, consider any $p, q \in \mathsf{Perm}(k)$ with $p \neq q$, and let $r := u_p v_q$. Choose the largest $i$ for which $p(i) \neq q(i)$. As $p(j) = q(j)$ for all $j > i$, $v_p$ and $v_q$ have the common prefix $\dashv_{x_{p(k)}} \mathsf{a} \cdots \dashv_{x_{p(i+1)}} \mathsf{a}$, and the leftmost letter where the two ref-words disagree is $\dashv_{x_{p(i)}}$ (in $v_p$) and $\dashv_{x_{q(i)}}$ (in $v_q$). In $u_p$, $x_{p(i)}$ is opened after $x_{q(i)}$ is opened (hence, it is closed in $v_p$ before $x_{q(i)}$ is closed. But in $v_q$, $x_{q(i)}$ is closed before $x_{p(i)}$, which means that while $u_p v_q$ is a valid ref-word, it defines an $X_k$-tuple that is not hierarchical, which means that it cannot correspond to any ref-word that

is defined by a vstk-automaton (in particular not by $A_k$). Hence, as $A$ and $A_k$ are equivalent, $u_p v_q \notin \mathsf{Ref}(A)$ must hold. This allows us to conclude that $A$ has at least $|\mathsf{Perm}(k)| = k!$ states. ◀

This lower bound is not optimal: By modifying the construction for the upper bound for vset-automata from Proposition 12, it is possible to give not only an upper bound on the number of states of vset-automata that simulate vstk-automata, but also a matching lower bound.[1]

## A.7 Proof of Lemma 20

**Proof.** We show this by reduction from the problem of checking whether $[\![A]\!](\varepsilon) \neq \emptyset$, which is NP-complete according to Lemma 11. Let $A \in \mathsf{VA}$, and assume that we can construct in polynomial time a $\varphi \in \mathsf{SpLog}(\mathsf{W})$ that realizes $A$. Then $[\![A]\!](\varepsilon) \neq \emptyset$ holds if and only if there is a $\sigma$ with $\sigma \models \varphi$ and $\sigma(\mathsf{W}) = \varepsilon$. As $\sigma$ maps every variable in $\varphi$ to a subword of $\sigma(\mathsf{W})$, $\sigma(x) = \varepsilon$ has to hold for all $x \in \mathsf{free}(\varphi)$. The same holds for all variables that are introduced with existential quantifiers. Hence, $\sigma \models \varphi$ if and only if $\sigma_\varepsilon \models \varphi$, where the substitution $\sigma_\varepsilon$ is defined by $\sigma_\varepsilon(x) := \varepsilon$ for all $x \in \Xi$.

Whether this holds can be easily verified by rewriting $\varphi$ into a Boolean expression over 1 and 0: Every equation $\mathsf{W} = \eta_R$ is replaced with 1 if $\sigma_\varepsilon(\eta_R) = \varepsilon$, and with 0 if $\sigma_\varepsilon(\eta_R) \neq \varepsilon$. Likewise, every constraint $\mathsf{C}_A(x)$ is replaced with 1 if $\varepsilon \in \mathcal{R}(A)$, and 0 if $\varepsilon \notin \mathcal{R}(A)$ (as $A$ is an NFA, this can be checked in polynomial time). Finally, all existential quantifiers are removed. This results in a Boolean expression (consisting of 0, 1, $\wedge$ and $\vee$), which we just need to evaluate. If the result is 1, we know that $[\![A]\!](\varepsilon) \neq \emptyset$; if it is 0, $[\![A]\!](\varepsilon) = \emptyset$ holds.

All this is possible in polynomial time. Hence, if a polynomial time conversion from $\mathsf{VA}_{\mathsf{set}}$ or $\mathsf{VA}_{\mathsf{stk}}$ to SpLog exists, $\mathsf{P} = \mathsf{NP}$ follows. ◀

## A.8 Proof of Theorem 21

As this proof is comparatively large, it has been split up into multiple section. The conversions from SpLog to various spanner representations can be found in Section A.8.1, while the conversions from these representations to SpLog can be found in Section A.8.2.

### A.8.1 From SpLog to Spanner Representations

As the proof is basically identical for all three types of primitive spanner representations (RGX, $\mathsf{VA}_{\mathsf{set}}$, and $\mathsf{VA}_{\mathsf{stk}}$), we consider all three at the same time.

[1] For the upper bound, the constructed vset-automaton stores a set $O \subseteq X_k$ of variables that have been opened (and might be still open), and a vector $\vec{v}$ of variables that are currently open (i. e., the elements of $\vec{v}$ are a subset of $O$). If $M_k$ is defined to by the set of all such $(O, \vec{v})$ for $k$ variables, we can turn each vstk-automaton with $n$ states and $k$ variables into an equivalent vstk-automaton with $nf(k)$ states, where $f(k) := |M_k|$ (the elements of $M_k$ contain enough information to simulate how the vset-automaton closes variables). Furthermore, we can construct appropriate tuples for each element of $M_k$ to use Lemma 39 to show that this bound can be reached. It is easy to see that $f(k) = \sum_{i=0}^{k} \left( \binom{k}{i} \sum_{j=0}^{i} \frac{i!}{(i-j)!} \right) = \sum_{i=0}^{k} \sum_{j=0}^{i} \frac{k!}{(k-i)!(i-j)!}$, as there are $\binom{k}{i}$ possible choices of a set $O$ of size $i$, each of which gives $\frac{i!}{(i-j)!}$ possible choices of a vector $\vec{v}$ with $j$ elements from $O$. As the reasoning for this is more cumbersome than for Proposition 13, and as $k!$ is a more insightful lower bound than this $f(k)$, the author considers this observation just sufficiently interesting to warrant a very long footnote in this Appendix.

**Word equations:** Consider the word equation $\eta := (\mathsf{W} = \eta_R)$ with $\eta_R = \eta_1 \cdots \eta_n$, $n \geq 0$, and $\eta_i \in (\Sigma \cup \Xi) - \{\mathsf{W}\}$ for $1 \leq i \leq n$. Assume $\mathsf{var}(\eta_R) = \{x_1, \ldots, x_k\}$ for some $k \geq 0$. If $n = 0$ (and $\eta_R = \varepsilon$), we output the functional regex formula $\varepsilon$ (or an equivalent automaton).

Otherwise, assume that we want to construct a regex formula (the case for each of the automata representations proceeds analogously). We define the $\alpha := \alpha_1 \cdots \alpha_n$ as follows: If $\eta_i \in \Sigma$, then $\alpha_i := \eta_i$. Otherwise, $\eta_i = x$ with $x \in \Xi$. We distinguish two subcases: If $|\eta_1 \cdots \eta_{i-1}|_x = 0$, then $i$ is the leftmost occurrence of $x$ in $\eta_R$, and we define $\alpha_i := x\{\Sigma^*\}$, and $l_x := i$. Otherwise, let $\alpha_i := x^{(i)}\{\Sigma^*\}$.

Next, define $\rho := \pi_Y S\alpha$, where $Y := \mathsf{var}(\eta_R)$, and $S$ is a sequence of selections $\zeta^{=}_{x,x^{(j)}}$ for each $x \in \mathsf{var}(\eta_R)$ and each $j > l_x$ with $\eta_j = x$.

Clearly, $\rho$ can be computed in polynomial time. Note that the regex formula $\alpha$ is functional, as each occurrence of $x \in \mathsf{var}(\eta_R)$ is converted into a distinct variable $x$ or $x^{(i)}$. In addition to this, we can turn $\alpha$ into a functional vset- or vstk-automaton. Furthermore, the projection $\pi_Y$ ensures that $\mathsf{SVars}(\rho) = \mathsf{var}(\eta_R) = \mathsf{free}(\eta) - \{\mathsf{W}\}$.

In order to see that the construction is correct, we first assume that there is a $\sigma$ with $\sigma \models \eta$; i. e., $\sigma(\mathsf{W}) = \sigma(\eta_R)$. Let $w := \sigma(\mathsf{W})$, and $w_i := \sigma(x_i)$ for $1 \leq i \leq k$.

We now have to construct a $\mu \in [\![\rho]\!](w)$ with $w_{\mu(x_i)} = w_i$ for $1 \leq i \leq k$. In order to do so, consider the ref-word $r = r_1 \cdots r_n$, which we define in the following way: For $1 \leq i \leq k$, if $\eta_i \in \Sigma$, let $r_i = \eta_i$. Otherwise, $\eta_i = x$ for some $x \in \Xi$. If $i = l_x$, let $r_i := \vdash_x \sigma(x) \dashv_x$. Otherwise, let $r_i := \vdash_{x^{(i)}} \sigma(x) \dashv_{x^{(i)}}$. As $\sigma(\mathsf{W}) = \sigma(\eta_R) = w$, $\mathsf{clr}(r) = w$ must hold. Hence, and as $r$ follows the same construction principle as $\alpha$, $r \in \mathsf{Ref}(\alpha, w)$. Furthermore, $w_{\mu^r(x)} = w_{\mu^r(x^{(i)})} = \sigma(x)$ holds for all $x \in \mathsf{var}(\eta)$ and all $l_x < i$ with $\eta_i = x$. Hence, $\mu^R \in [\![S\alpha]\!](w)$. This implies $\mu \in [\![\rho]\!](w)$ for $\mu := \mu^r|_Y$, which concludes this direction of the proof.

For the opposite direction, assume that $\mu \in [\![\rho]\!](w)$ for some $w \in \Sigma^*$. By definition, there exists an $r \in \mathsf{Ref}(\alpha, w)$ with $\mu = \mu^r|_Y$. The construction of $\alpha$ allows us to factorize $r$ into $r = r_1 \cdots r_n$, where for each $1 \leq i \leq n$, one of three cases holds:

1. $r_i \in \Sigma$ and $r_i = \eta_i$,
2. $r_i = \vdash_x u_i \dashv_x$, with $u_i \in \Sigma^*$, $x \in \Xi$, and $i = l_x$,
3. $r_i = \vdash_{x^{(i)}} u_i \dashv_x$, with $u_i \in \Sigma^*$, $x \in \Xi$, and $i > l_x$.

Furthermore, as $\mu \in [\![S\alpha]\!](w)$, $u_{l_x} = u_i$ holds for all $x \in \Xi$ and all $i > l_x$ with $\eta_i = x$. Hence, if we define a substitution $\sigma$ by $\sigma(\mathsf{W}) := w$, and $\sigma(x) := u_{l_x}$ for all $x \in \mathsf{var}(\eta_R)$, we obtain $\sigma(\eta_R) = w = \sigma(\mathsf{W})$, and conclude $\sigma \models \eta$.

**Constraint symbols:** Let $\psi := (\varphi \wedge \mathsf{C}_A(x))$ (as $\mathsf{SpLog}$ formulas are safe, constraint symbols occur only as part of formulas $\varphi \wedge \mathsf{C}_A(x)$, with $x \in \mathsf{free}(\varphi)$). Let $\rho_\varphi$ be an appropriate spanner representation that realizes $\varphi$, let $x^T$ be a new variable. If $A$ is a regular expression and our goal is to construct a regex formula, let $\rho_A := \Sigma^* \cdot x_T\{A\} \cdot \Sigma^*$. (If we want to construct a v-automaton, or $A$ is an NFA, $\rho_A$ is defined accordingly.) Now, let $\rho := \pi_Y \zeta^{=}_{x,x^T}(\rho_\varphi \times \rho_A)$, where $Y := \mathsf{free}(\varphi) - \{\mathsf{W}\}$. In order to see that $\rho$ realizes $\psi$, observe that for all $w$, $\mu \in [\![\rho]\!](w)$ holds if and only if both $\mu \in [\![\rho_\varphi]\!]$ and $w_{\mu(x)} = w_{\mu(x^T)} \in \mathcal{R}(A)$.

**Disjunctions:** Let $\psi := (\varphi_1 \vee \varphi_2)$, where $\varphi_1, \varphi_2 \in \mathsf{SpLog}(\mathsf{W})$ are realized by spanner representations $\rho_1$ and $\rho_2$. As $\psi$ is safe, $\mathsf{free}(\varphi_1) = \mathsf{free}(\varphi_2)$ holds, which implies $\mathsf{SVars}(\rho_1) = \mathsf{SVars}(\rho_2)$. Hence, we can define $\rho := (\rho_1 \cup \rho_2)$. We conclude that $\rho$ realizes $\psi$ directly from the definitions.

**Conjunctions:** Let $\psi := (\varphi_1 \wedge \varphi_2)$, where $\varphi_1, \varphi_2 \in \mathsf{SpLog}(\mathsf{W})$ are realized by spanner representations $\rho_1$ and $\rho_2$. Let $Y := (\mathsf{SVars}(\varphi_1) \cap \mathsf{SVars}(\varphi_2)) - \{\mathsf{W}\}$, and let $\hat{\rho}_2$ be the spanner representation that is obtained from $\rho_2$ by renaming each $x \in Y$ to a new variable $x^T$. Now define $\rho := \pi_Y S(\rho_1 \times \hat{\rho}_2)$, where $S$ is a sequence of selections $\zeta^=_{x,x^T}$ for each $x \in Y$. (The renaming allows us to use $\times$ instead of $\bowtie$). Due to the selections, we observe that $\mu \in [\![\rho]\!](w)$ holds if and only if, firstly, $\mu \in [\![\rho_1]\!](w)$ and, secondly, there is a $\hat{\mu}_2 \in [\![\hat{\rho}_2]\!](w)$ such that $w_{\mu(x)} = w_{\hat{\mu}_2(x^T)}$ for all $x \in Y$. Define $\mu_2$ by $\mu_2(x) := \hat{\mu}_2(x^T)$ for each $x \in Y$. Then $\mu_2 \in [\![\rho_2]\!](w)$ holds if and only if $\hat{\mu}_2 \in [\![\hat{\rho}_2]\!](w)$. Now it is easily seen that $\rho$ realizes $\psi$.

**Existential quantifiers:** Let $\psi := (\exists x\colon \varphi)$, with $\varphi \in \mathsf{SpLog}(\mathsf{W})$, $x \in \mathsf{free}(\varphi) - \{\mathsf{W}\}$, and let $\varphi$ be realized by some spanner representation $\rho_\varphi$. Then we simply define $\rho := \pi_Y \rho_\varphi$, with $Y := \mathsf{free}(\varphi) - \{\mathsf{W}, x\}$. Again, we can conclude that $\rho$ realizes $\varphi$ directly from the definitions.

## A.8.2 From Spanner Representations to SpLog

The conversions proceed as follows:

1. First, the primitive spanner representations are converted into SpLog-formulas:
   a. For regex formulas, see Section A.8.2.1.
   b. For vset-automata, see Section A.8.2.2.
   c. For vstk-automata, see Section A.8.2.3.
2. Then these formulas are combined according to the spanner operators, see Section A.8.2.4.

### A.8.2.1 Conversion of Functional Regex Formulas

The main part of this construction was already presented by Freydenberger and Holldack in the proof of Theorem 27 in [14]. For the convenience of the reviewers, that proof is included in Section B.1 in the second Appendix.

There, the proof of Claim 2 establishes that there are polynomial time conversions from $\mathsf{RGX}^{\mathsf{core}}$ to $\mathsf{EC}^{\mathsf{reg}}$. As we want to construct a $\mathsf{SpLog}_{\mathsf{rx}}$-formula, we need to make the following changes and observations:

1. $x_w$ is the main variable of the constructed $\mathsf{SpLog}_{\mathsf{rx}}$-formulas.
2. In case 1 ($\rho$ is a proper regular expression), we instead define $\varphi_\rho(x_w) := \exists x\colon (x_w = x \wedge \mathsf{C}_\rho(x))$. This formula is equivalent, but safe.
3. Case 2a remains unchanged: If $\varphi_{\rho_1}, \varphi_{\rho_2} \in \mathsf{SpLog}_{\mathsf{rx}}(x_W)$, so is $\varphi_\rho$.
4. Case 2b also remains unchanged. Note that Lemma 17 allows us to use $\mathsf{SpLog}_{\mathsf{rx}}$-formulas with other main variables in the definition of this formula, and that this does not cause complexity issues (see the discussion after that lemma).
5. Case 2c only requires the cosmetic change of replacing $(x_1^C = x_w)$ with $(x_w = x_1^C)$, and the mention that we use Lemma 17.

Apart from this, the proof proceeds as in Section B.1. Hence, we conclude that there is a polynomial time conversion from RGX to $\mathsf{SpLog}_{\mathsf{rx}}$.

### A.8.2.2 Conversion of vset-Automata

The construction for vset-automata is more involved than for regex formulas. The main reason for this is that the latter are restricted to functional regex formulas, which ensure syntactically that every variable is assigned exactly one value. In contrast to this, vset-automata ensure this assignment in their behavior (in the original semantics from [12], this is ensured in the definition of accepting runs; while our ref-word definition ensures this through the definition of Ref).

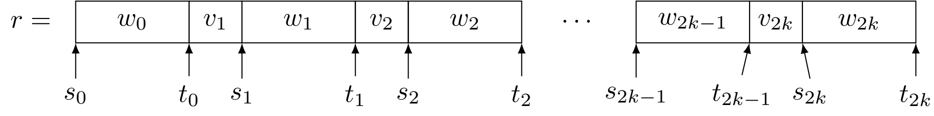



**Figure 3** A graphical representation of the factorization of $r \in \mathsf{Ref}(A, w)$, which is used in the proof of Theorem 21, Section A.8.2.2. The $s_i$ and $t_i$ denote the states before and after processing the $w_i$, while the $v_i$ denote variable operations.

While one could encode all possible combinations of how variables overlap, this would result in a formula that is of exponential size (if the number of variables is unbounded). As our goal is to construct a formula in polynomial time (and, hence, also polynomial size), we choose a more refined approach. Let $A = (Q, q_0, q_f, \delta)$ be a vset-automaton, and let $\mathsf{SVars}\,(A) = \{x_1, \dots, x_k\}$, $k \geq 0$.

We now make some observations that form the fundament the construction: For every $w \in \Sigma^*$, every $r \in \mathsf{Ref}(A, w)$ has a unique factorization

$$r = w_0 \cdot v_1 \cdot w_1 \cdot v_2 \cdots w_{2k-1} \cdot v_{2k} \cdot w_{2k},$$

with $w_i \in \Sigma^*$ and $v_i \in \{\vdash_{x_j}, \dashv_{x_j} \mid 1 \leq j \leq k\}$. Then $w = w_0 \cdot w_1 \cdots w_{2k}$, while the $v_i$ describe the used variable operations (opening or closing variables). Furthermore, there exist states $s_0, \dots, s_{2k}, t_0, \dots, t_{2k} \in Q$ such that the following holds:

1. $t_i \in \delta(s_i, w_i)$ for each $0 \leq i \leq 2k$,
2. $s_{j+1} \in \delta(t_j, v_j)$ for each $1 \leq j \leq 2k$,
3. $s_0 = q_0$,
4. $t_{2k} = q_f$.

A graphical representation of this can be found in Figure 3.

The basic idea of the construction is the use of variables in the formula in order to store information like the states, and when variables are opened and closed. Two central limitations of SpLog are that each variable has to be a subword of W, and that it is a purely positive theory. Nonetheless, some limited synchronisation is possible in SpLog: For each information that has to be stored (e. g., for each state $s_i$), we define a group of variables that represents the (finitely many) possible choices (e. g. variables $s_i^q$ for all $q \in Q$), and ensure that for every satisfying $\sigma$, exactly one of these variables is mapped to the first letter of W, while all others are mapped to $\varepsilon$.

Our goal is now to construct a SpLog-formula that realizes $A$ on all $w \in \Sigma^+$ (the case of $w = \varepsilon$ is ignored, as we want to construct a polynomial time conversion modulo $\varepsilon$).

As mentioned above, the construction uses various sets of variables, where in each set, exactly one shall be mapped to the first letter of $w$, while all others are mapped to $\varepsilon$. This allows us to synchronize different parts of the equation, and to store non-deterministic decisions, like the assigned states. We use the following sets of variables:

1. For $0 \leq i \leq 2k$, $S_i := \{s_i^q \mid q \in Q\}$, where $s_i^q = \hat{a}$ represents $s_i = q$,
2. For $0 \leq i \leq 2k$, $T_i := \{t_i^q \mid q \in Q\}$, where $t_i^q = \hat{a}$ represents $t_i = q$,
3. For $1 \leq i \leq k$, $O_i := \{o_i^j \mid 1 \leq j \leq 2k\}$, where $o_i^j = \hat{a}$ represents $v_j = \vdash_{x_i}$,
4. For $1 \leq i \leq k$, $C_i := \{c_i^j \mid 1 \leq j \leq 2k\}$, where $c_i^j = \hat{a}$ represents $v_j = \dashv_{x_i}$.

In order to manage these variables, we heavily rely on four types of auxiliary formulas. We begin with the formulas that handle the allocation of the states $s_i$ and $t_i$. For $0 \leq i \leq 2k$,

$q \in Q$, let

$$\varphi_{s,i}^q := (s_i^q = \hat{a}) \wedge \bigwedge_{\substack{p \in Q,\\ p \neq q}} (s_i^p = \varepsilon).$$

We define $\varphi_{t,i}^q$ analogously. On an intuitive level, $\varphi_{s,i}^q$ represents that $s_i = q$. Note that $\mathsf{free}(\varphi_{s,i}^q) = S_i \cup \{\mathsf{W}, \hat{a}\}$ and $\mathsf{free}(\varphi_{t,i}^q) = T_i \cup \{\mathsf{W}, \hat{a}\}$, as we implicitly assume the formulas to be $\mathsf{SpLog}(\mathsf{W})$-formulas (see Lemma 17, and the discussion thereafter). In fact, this definition of $\varphi_{s,i}^q$ is to be understood as a notational shorthand for the equivalent (but less readable) $\mathsf{SpLog}(w)$-formula

$$\varphi_{s,i}^q = (\exists \hat{w}\colon\ (\mathsf{W} = s_i^q \hat{w}) \wedge (\mathsf{W} = \hat{a}\hat{w})) \wedge \bigwedge_{\substack{p \in Q,\\ p \neq q}} (\exists \hat{w}\colon\ (\mathsf{W} = s_i^p \hat{w}) \wedge (\mathsf{W} = \hat{w}))\,.$$

This equivalence only holds because we shall ensure that $\hat{a}$ refers to the first letter of $\mathsf{W}$. This shall be the task of a formula $\varphi_{\hat{a}}$ that is defined later. Further down, the fact that the set of free variables depends only on $i$, and not on $q$, shall allow us to use these formulas in disjunctions.

In order to handle the variable operations, for $1 \leq i \leq k$ and $1 \leq j \leq 2k$, we define

$$\varphi_{o,i}^j := (o_i^j = \hat{a}) \wedge \bigwedge_{\substack{1 \leq l \leq 2k,\\ l \neq j}} (o_i^l = \varepsilon),$$

and $\varphi_{c,i}^j$ is defined analogously. Like our formulas for the states $s_i$ and $t_i$, $\varphi_{o,i}^j$ and $\varphi_{c,i}^j$ represent $v_j = \vdash_{x_i}$ and $v_j = \dashv_{x_i}$, respectively. Again, we observe $\mathsf{free}(\varphi_{o,i}^j) = O_i \cup \{\mathsf{W}, \hat{a}\}$, and $\mathsf{free}(\varphi_{c,i}^j) = C_i \cup \{\mathsf{W}, \hat{a}\}$, which we shall also use to construct disjunctions.

While the formulas $\varphi_{o,i}^j$, $\varphi_{c,i}^j$ allow us to check where a variable $x_i$ is opened or closed, we also need formulas that express the opposite direction, i. e., which variable $x_j$ is opened or closed in some operation $v_i$. To this end, we define for $1 \leq i \leq 2k$ and $1 \leq j \leq k$ the formulas

$$\varphi_{v,i}^{\vdash,j} := \left((o_j^i = \hat{a}) \wedge (c_j^i = \varepsilon)\right) \wedge \bigwedge_{\substack{1 \leq l \leq k,\\ l \neq j}} \left((o_l^i = \varepsilon) \wedge (c_l^i = \varepsilon)\right),$$

$$\varphi_{v,i}^{\dashv,j} := \left((o_j^i = \varepsilon) \wedge (c_j^i = \hat{a})\right) \wedge \bigwedge_{\substack{1 \leq l \leq k,\\ l \neq j}} \left((o_l^i = \varepsilon) \wedge (c_l^i = \varepsilon)\right).$$

Like $\varphi_{o,j}^i$, $\varphi_{v,i}^{\vdash,j}$ expresses that $v_i = \vdash_{x_j}$, and $\varphi_{c,j}^i$ and $\varphi_{v,i}^{\dashv,j}$ both express $v_i = \dashv_{x_j}$. But note that $\mathsf{free}(\varphi_{v,i}^{\vdash,j}) = \mathsf{free}(\varphi_{v,i}^{\dashv,j}) = \{\mathsf{W}, \hat{a}\} \cup \{o_j^i, c_j^i \mid 1 \leq j \leq k\}$. Hence, these new formulas can be used in disjunctions where the variable operation is fixed (instead of the variable).

We now define the formula

$$\varphi := \exists \vec{v}\colon \varphi_{\hat{a}} \wedge \varphi_{fact} \wedge \varphi_{start} \wedge \varphi_{acc} \wedge \varphi_{span} \wedge \varphi_{trans}^{w} \wedge \varphi_{trans}^{v},$$

where the sequence of variables $\vec{v}$ is an arbitrary ordering of the variable set

$$V := \{\mathsf{a}, w_0, w_1, \ldots, w_{2k}\} \cup \bigcup_{i=0}^{2k} S_i \cup \bigcup_{i=0}^{2k} T_i \cup \bigcup_{i=1}^{k} O_i \cup \bigcup_{i=1}^{k} C_i,$$

and the subformulas of $\varphi$ are defined as follows:

- $\varphi_{\hat{a}} := \exists \hat{w}\colon (\mathsf{W} = \hat{a} \cdot \hat{w})$. This formula stores the first letter of $w$ in $\hat{a}$, the special variable that will be used to synchronize various subformulas.

- $\varphi_{fact} := (\mathsf{W} = w_0 \cdot w_1 \cdots w_{2k})$. This formula factorizes $w$ into $w = w_0 \cdot w_1 \cdots w_{2k}$.
- $\varphi_{start} := \varphi^{q_0}_{s,0}$. This formula ensures $s_0 = q_0$
- $\varphi_{acc} := \varphi^{q_f}_{t,2k}$. This expresses $t_{2k} = q_f$.
-

$$\varphi_{span} := \bigwedge_{i=1}^{k} \bigvee_{j=1}^{2k-1} \bigvee_{l=j+1}^{2k} \left(\varphi^{j}_{o,i} \land \varphi^{l}_{c,i} \land \varphi_{fact} \land \left(x^P_i = w_0 \cdots w_{j-1}\right) \land \left(x^C_i = w_j \cdots w_{l-1}\right)\right)$$

To every $x_i$, this formula assigns a range between $v_j$ and $v_l$, by setting $v_j = \vdash_{x_i}$ and $v_l = \dashv_{x_i}$ with $l > j$, as well as $x^P_i = w_0 \cdots w_{j-1}$, $x^C_i = w_j \cdots w_{l-1}$.
To see that $\varphi_{span}$ is safe, note that or each $i$, the formula consists of a disjunction of formulas, each of which has free variables $O_i \cup C_i \cup \{\mathsf{W}, \hat{a}, w_0, \dots, w_{2k}, x^P_i, x^C_i\}$.
- $\varphi^{w}_{trans}$ has to check whether each $w_i$ corresponds to a path from $s_i$ to $t_i$ in $A$, where each edge along the path is labeled only with letters from $\Sigma$. In order to define this, for each pair $p, q \in Q$, we define an NFA $A_{p,q} := (Q, \delta_{p,q}, p, \{q\})$, where $\delta_{p,q}$ is the restriction of $\delta$ to $Q \times \Sigma \to Q$. In other words, for all $\hat{q} \in Q$ and all $x \in (\Sigma \cup \Gamma)$,

$$\delta_{p,q}(\hat{q}, x) := \begin{cases} \delta(\hat{q}, x) & \text{if } x \in \Sigma, \\ \emptyset & \text{if } x \in \Gamma. \end{cases}$$

Hence, each $A_{p,q}$ is the NFA over $\Sigma$ that simulates $A$ when starting in $p$, accepting in $q$, and processing only edges with labels from $\Sigma$ (while ignoring labels from $\Gamma$).
Then we define

$$\varphi^{w}_{trans} := \bigwedge_{i=0}^{2k} \bigvee_{p,q \in Q} \left(\varphi^{p}_{s,i} \land \varphi^{q}_{t,i} \land (w_i \sqsubseteq \mathsf{W}) \land \mathsf{C}_{A_{p,q}}(w_i)\right),$$

where we use $w_i \sqsubseteq \mathsf{W}$ as shorthand for $\exists \hat{w}_1, \hat{w}_2 \colon (\mathsf{W} = \hat{w}_1 \cdot w_i \cdot \hat{w}_2)$. (This has to be included, otherwise, we could not use $\mathsf{C}_{A_{p,q}}(w_i)$ inside the conjunction.) Again, it is easily seen that the formula is safe, as for each $i$, the disjunction ranges over subformulas that have the free variables $\{\mathsf{W}, \hat{a}, w_i\} \cup S_i \cup T_i$.
Now, $\varphi^{w}_{trans}$ states that for each $0 \leq i \leq 2k$, $w_i \in \mathcal{L}(A_{s_i,t_i})$, which is equivalent to $t_i \in \delta(s_i, w_i)$.
- $\varphi^{v}_{trans}$ has to check that for each $v_i$, $s_i \in \delta(t_{i-1}, v_i)$. We define $\varphi^{v}_{trans}$ as

$$\bigwedge_{i=1}^{2k} \bigvee_{j=1}^{k} \left( \left( \varphi^{\vdash,j}_{v,i} \land \bigvee_{\substack{p \in Q,\\ q \in \delta(p, \vdash_{x_j})}} (\varphi^{p}_{t,i-1} \land \varphi^{q}_{s,i}) \right) \lor \left( \varphi^{\dashv,j}_{v,i} \land \bigvee_{\substack{p \in Q,\\ q \in \delta(p, \dashv_{x_j})}} (\varphi^{p}_{t,i-1} \land \varphi^{q}_{s,i}) \right) \right)$$

Here, $\varphi^{v}_{trans}$ considers each $v_i$, finds the (unique) $j$ with $v_i = \{\vdash_{x_i}, \dashv_{x_i}\}$, and ensures that $s_i \in \delta(t_{i-1}, \vdash_{x_j})$. To see that the formulas are safe, first recall that for the used auxiliary formulas, the set of free variables depends only on $i$, not on $j$, $p$, or $q$. Also recall that $\mathsf{free}(\varphi^{\vdash,j}_{v,i}) = \mathsf{free}(\varphi^{\dashv,j}_{v,i})$ holds by definition.

**Correctness:** In order to see the correctness of this construction, recall the explanations that are provided with each subformula. First, we examine why every $\sigma \in [\![\varphi]\!]$ corresponds to an $r \in \mathsf{Ref}(A, \sigma(\mathsf{W}))$. By $\varphi_{fact}$, we have $\sigma(\mathsf{W}) = \sigma(w_0) \cdots \sigma(w_{2k})$. Furthermore, $\varphi_{span}$ and $\varphi^{v}_{trans}$ ensure that the $v_i$ are valid for a word from $\mathsf{Ref}(A, \sigma(\mathsf{W}))$: Due $\varphi^{v}_{trans}$, every $v_i$ with $1 \leq i \leq 2k$ is assigned exactly one value from the set $\{\vdash_{x_1}, \dots, \vdash_{x_k}, \dashv_{x_1}, \dots, \dashv_{x_k}\}$;

and due to $\varphi_{span}$, for every $1 \leq i \leq k$, there exist exactly one $j$ and one $l$, $1 \leq j < l \leq k$, such that $v_j = \vdash_{x_i}$ and $v_l = \dashv_{x_i}$. Next, we check that $r$ corresponds to an accepting run of $A$: $\varphi_{start}$ and $\varphi_{acc}$ ensure $s_0 = q_0$ and $t_{2k} = q_f$, respectively. For $0 \leq i \leq 2k$, $\varphi^w_{trans}$ guarantees $t_i \in \delta(q_i, \sigma(w_i))$, while $\varphi^v_{trans}$ enforces $s_i \in \delta(t_{i-1}, v_i)$ for $1 \leq i \leq 2k$. This allows us to conclude that $\sigma$ encodes an $r \in \mathsf{Ref}(A, \sigma(\mathsf{W}))$. Finally, $\varphi_{span}$ also ensures that the span variables $x_i^P, x_i^C$ have the correct contents.

For the other direction, assume that $r \in \mathsf{Ref}(A, w)$. As explained above, $r$ has a unique factorization $r = w_0 v_1 w_1 \cdots v_{2k} w_{2k}$, from which we can directly derive a $\sigma \in [\![\varphi]\!]$.

**Complexity:** As $\varphi$ is directly derived from the structure of $A$ (with some reachability checking to determine the NFAs $A_{p,q}$), in oder to prove that $\varphi$ can be computed in polynomial time, it suffices to show that the size of $\varphi$ is polynomial in the size of $A$. Let $n := |Q|$, and recall that $k = |\mathsf{SVars}(A)|$. First, we observe that $|V| \in O(k^2 + kn)$ (as each of the sets $S_i, T_i$ has $O(n)$ elements, while each of the sets $C_i, O_i$ has $O(k)$ elements, and each type of sets occurs $O(k)$ times). More importantly, each of the auxiliary formulas $\varphi^q_{s,i}$, $\varphi^q_{t,i}$, $\varphi^j_{o,i}$, $\varphi^j_{c,i}$, $\varphi^{\vdash,j}_{v,i}$, and $\varphi^{\dashv,j}_{v,i}$ is of size $O(k)$. This allows us to make the following observations:

- $\varphi_{\hat{a}}$ has size $O(1)$.
- $\varphi_{fact}$ has size $O(k)$.
- $\varphi_{start}$ has size $O(k)$.
- $\varphi_{acc}$ has size $O(kn)$, the disjunction can have $|F| \in O(n)$ elements, each of size $O(k)$.
- $\varphi_{span}$ has size $O(k^4)$, as the junctors range over $O(k^3)$ suformulas, each of size $O(k)$.
- $\varphi^w_{trans}$ has size $O(k^2 n^3)$, as the junctors range over $O(kn^2)$ subformulas, each of size $O(kn)$. (If we assume that each $A_{p,q}$ is specified explicitly.)
- $\varphi^v_{trans}$ has size $O(k^3 n^2)$. The two outer junctors range over $O(k^2)$ subformulas. Each of these subformulas is a disjunction, where both sides have the same size, which is $O(k + kn^2)$. Hence, multiplication yields $O(k^3 n^2)$.

We arrive at a total size of $O(k^4 + k^3 n^2 + k^2 n^3)$, which is polynomial in the size of $A$.

#### A.8.2.3 Conversion of vstk-Automata

The construction for vstk-automata is very similar to the construction for vset-automata (see Section A.8.2.2). But as vstk-automata do not close variables explicitly, we need to extend the constructed formula. Let $A = (Q, q_0, q_f, \delta)$ be a vstk-automaton with $\mathsf{SVars}(A) = \{x_1, \ldots, x_k\}$, $k \geq 0$.

For every $w \in \Sigma^*$, every $\hat{r} \in \mathsf{Ref}(A, w)$ can be rewritten into an $r \in (\Sigma \cup \Gamma)^*$, such that $\mu^r = \mu^{\hat{r}}$, by replacing each $\dashv$ with an appropriate $\dashv_{x_i}$. Then $r$ has the same unique factorization $r = w_0 \cdot v_1 \cdot w_1 \cdot v_2 \cdots w_{2k-1} \cdot v_{2k} \cdot w_{2k}$, as in Section A.8.2.2. Hence, in order to be able to use the construction from the vset-automata case, we only need to ensure that variables are closed in the correct order.

We construct $\varphi := \exists \vec{v} \colon \varphi_{\hat{a}} \wedge \varphi_{fact} \wedge \varphi_{start} \wedge \varphi_{acc} \wedge \varphi_{span} \wedge \varphi^w_{trans} \wedge \varphi^v_{trans} \wedge \varphi_{stack}$, where all formulas are defined as in Section A.8.2.2, with two differences:

- $\varphi^v_{trans}$ still has to check that for each $v_i$, $s_i \in \delta(t_{i-1}, v_i)$, but we cannot use $\dashv_{x_i}$. Instead, we define $\varphi^v_{trans}$ as

$$\bigwedge_{i=1}^{2k} \bigvee_{j=1}^{k} \left( \left( \varphi^{\vdash,j}_{v,i} \wedge \bigvee_{\substack{p \in Q, \\ q \in \delta(p, \vdash_{x_j})}} (\varphi^p_{t,i-1} \wedge \varphi^q_{s,i}) \right) \vee \left( \varphi^{\dashv,j}_{v,i} \wedge \bigvee_{\substack{p \in Q, \\ q \in \delta(p, \dashv)}} (\varphi^p_{t,i-1} \wedge \varphi^q_{s,i}) \right) \right)$$

Hence, $\varphi^v_{trans}$ can interpret each $\dashv$ as any $\dashv_{x_i}$. This does not ensure that variables are closed in the correct order (this is done by $\varphi_{stack}$).

- $\varphi_{stack}$ states that each closing operator closes the most recent open variable. To this end, we define $\varphi_{stack}$ as

$$\bigwedge_{1\leq i<k}\ \bigwedge_{i<j\leq k}\ \bigvee_{\substack{1\leq l_1<l_2,\\ l_2<l_3<l_4\leq 2k}} \Big( (\varphi^{l_1}_{o,i} \wedge \varphi^{l_2}_{o,j} \wedge \varphi^{l_3}_{c,j} \wedge \varphi^{l_4}_{c,i}) \vee (\varphi^{l_1}_{o,i} \wedge \varphi^{l_2}_{c,i} \wedge \varphi^{l_3}_{o,j} \wedge \varphi^{l_4}_{c,j}) \vee (\varphi^{l_1}_{o,j} \wedge \varphi^{l_2}_{o,i} \wedge \varphi^{l_3}_{c,i} \wedge \varphi^{l_4}_{c,j}) \vee (\varphi^{l_1}_{o,j} \wedge \varphi^{l_2}_{c,j} \wedge \varphi^{l_3}_{o,i} \wedge \varphi^{l_4}_{c,i}) \Big).$$

In order to understand this formula, let $o_i, c_i \in \{1, \dots, 2k\}$ such that $v_{o_i} = \vdash_{x_i}$, and $v_{c_i} = \dashv_{x_i}$, and define $o_j, c_j$ analogously for $x_j$. The four parts of the inner disjunction describe each possible combination how $x_i$ and $x_j$ can be opened and closed according to the rules of a vset-automaton: The first expresses $o_i < o_j < c_j < c_i$, the second $o_i < c_i < o_j < c_j$, and remaining two express the same for switched roles of $x_i$ and $x_j$. Hence, for any pair $i, j$, this ensures that if $x_j$ is opened while $x_i$ is open, $x_j$ has to be closed before $x_i$ can be closed (and vice versa). As $\varphi_{stack}$ expresses this for all pairs of variables in $\mathsf{SVars}(A)$, this ensures that all variables are closed correctly. The formula is safe, as for all fixed $i, j$, the disjunctions range over formulas with free variables $\{\mathsf{W}\} \cup O_i \cup O_j \cup C_i \cup C_j$.

The correctness of the construction follows immediately from our remarks on $\varphi_{stack}$, and from correctness of the construction from Section A.8.2.2. Regarding the complexity, we observe that $\varphi_{stack}$ is of size $O(k^7)$: There are $O(k^2)$ different combinations of $i$ and $j$. Each of these leads to $O(k^4)$ choices for $l_1$ to $l_4$, each of which requires a formula of size $O(k)$ (as we use a constant amount of auxiliary formulas of size $O(k)$). This leads to a total size of $O(k^7 + k^3n^2 + k^2n^3)$, which is larger than for vset-automata, but still polynomial in the size of $A$.

#### A.8.2.4 Putting The Parts Together

Here, we can directly use the construction from the proof of Theorem 27 in [14], see Section B.2. For all constructed formulas, it is easily seen that the resulting formulas are SpLog-formulas with main variable $x_w$. In the case of $\rho = \zeta^{=}_{\vec{x}}\hat{\rho}$, this is also a case where Lemma 17 is used.

## A.9 Proof of Corollary 22

**Proof.** These polynomial time conversions also imply an polynomial upper bound on the size of the computed representations. For the conversion of v-automata to SpLog-formulas, this size bound also holds if we omit the modulo $\varepsilon$, as for every $A \in \mathsf{VA}$, there are only two possible cases: Either $[\![A]\!](\varepsilon) = \emptyset$, or $[\![A]\!](\varepsilon) = \mu$, where $\mu(x) = [1, 1\rangle$ for all $x \in \mathsf{SVars}(A)$. In the latter case, we add this special case to the constructed formula. ◀

## A.10 Proof of Corollary 23

**Proof.** First, note that the proof of Theorem 21 constructs spanner representations use $\times$ with $\times$ instead of $\bowtie$, and that the constructed v-automata are functional. Hence, we can take a $\rho \in \mathsf{VA}^{\mathsf{core}}$, and convert it into a SpLog-formula $\varphi$, which is then converted into a $\hat{\rho} \in \mathsf{VA}_{\mathsf{set}}$ or $\hat{\rho} \in \mathsf{VA}_{\mathsf{stk}}$. We need one additional step, as the conversion to $\varphi$ doubles the number of variables (as every $x$ is turned into an $x^P$ and an $x^C$). In order to obtain $\rho_f$, we join $\hat{\rho}$ with $x^P\{\Sigma^*\} \cdot x^C\{\Sigma^*\} \cdot \Sigma^*$ for every $x$, and then projecting away the $x^P$. It is

also possible to solve this with $\times$ instead of $\bowtie$. To do so, for every $x$, we define a spanner $x_N\{\Sigma^*\}\cdot x\{\Sigma^*\}\cdot\Sigma^*$, which we $\times$ with $\hat{\rho}$ (and $x_N$ is a new variable). Before projecting $x_N$, $x^P$, and $x^C$ away, we select $\zeta^=_{x_N,x^P}$ and $\zeta^=_{x,x^C}$. ◀

## A.11 Proof of Corollary 24

**Proof.** We begin with the combined complexity: NP-hardness follows from the NP-hardness of evaluation of $\mathsf{RGX}^{\mathsf{core}}$, as shown in [14] (or, more elegantly, directly from the membership problem for pattern languages, that is used in that proof). For the upper bounds, we could refer to the corresponding upper bounds for $\mathsf{RGX}^{\mathsf{core}}$ in [14] and discuss the necessary modifications, but it is more convenient (and more elegant) to discuss this directly for SpLog.

The NP upper bound is due to the fact that, given $\varphi\in\mathsf{SpLog}(\mathsf{W})$ and $\sigma$, it suffices to guess a substitution for every variable that is existentially quantified in $\varphi$, and to verify this guess. As every variable has to be a subword of $\sigma(\mathsf{W})$, this is possible in polynomial time.

A similar reasoning proves the NL upper bound for data complexity: If $\varphi\in\mathsf{SpLog}(\mathsf{W})$ is fixed, we can use two pointers to represent each variable that occurs somewhere in $\varphi$ (by marking its first and its last letter in $\sigma(\mathsf{W})$. We can then guess a substitution for each variable, and verify the correctness of this substitution with a constant amount of additional pointers that track our way through $\varphi$. ◀

## A.12 Proof of Lemma 25

**Proof.** We first prove that every relation is selectable by core spanners if and only if it is SpLog-selectable. We only examine the "only if "-direction (the "if"-direction proceeds analogously). Assume that $R\subseteq(\Sigma^*)^k$ is selectable by core spanners. Let $\varphi\in\mathsf{SpLog}(\mathsf{W})$, choose $x_1,\dots,x_k\in\mathsf{free}(\varphi)-\{\mathsf{W}\}$, and define $\vec{x}=(x_1,\dots,x_k)$. Our goal is to show that there exists a $\varphi^R$ such that $\sigma\models\varphi$ holds if and only if $\sigma\models\varphi$ and $(\sigma(x_1),\dots,\sigma(x_k))\in R$. According to Theorem 21, there exists a $\rho\in\mathsf{RGX}^{\mathsf{core}}$ that realizes $\varphi$. More explicitly, this means that $\mathsf{SVars}(\rho)=\mathsf{free}(\varphi)-\{\mathsf{W}\}$, and for every substitution $\sigma$, $\sigma\models\varphi$ if and only if there exists a $\mu\in[\![\rho]\!](\sigma(\mathsf{W}))$ with $\sigma(\mathsf{W})_{\mu(x)}=\sigma(x)$ for all $x\in\mathsf{SVars}(\rho)$.

As $R$ is selectable by core spanners, there also exists a $\rho^R\in\mathsf{RGX}^{\mathsf{core}}$ with $[\![\rho^R]\!]=[\![\zeta^R_{\vec{x}}\rho]\!]$. Then $\mathsf{SVars}\left(\rho^R\right)=\mathsf{SVars}(\rho)$, and for all $w\in\Sigma^*$, $\mu\in[\![\rho^R]\!](w)$ holds if and only if $\mu\in[\![\rho]\!](w)$ and $(w_{\mu(x_1)},\dots,w_{\mu(x_k)})\in R$.

Hence, for all substitutions $\sigma$, there is a $\sigma\models\varphi$ with $(\sigma(x_1),\dots,\sigma(x_k))\in R$ if and only if there is a $\mu\in[\![\rho^R]\!](\sigma(\mathsf{W}))$ with $\sigma(\mathsf{W})_{\mu(x)}=\sigma(x)$ for all $x\in\mathsf{SVars}\left(\rho^R\right)$.

Again by Theorem 21, there exists a $\hat{\varphi}^R\in\mathsf{SpLog}$ that realizes $\rho^R$. Note that $\mathsf{free}(\hat{\varphi}^R)=\{\mathsf{W}\}\cup\{x^P,x^C\mid x\in\mathsf{free}(\varphi)-\{\mathsf{W}\}\}$. In order to clean this up, let $\tilde{\varphi}^R$ be obtained from $\hat{\varphi}^R$ by renaming each $x^C$ to $x$. Then define $\vec{p}$ as any ordering of the set $\{x^P\mid x\in\mathsf{free}(\tilde{\varphi}^R)\}$, and let $\varphi^R:=\exists\vec{p}\colon\tilde{\varphi}^R$. Then for every substitution $\sigma$, $\sigma\models\varphi^R$ if and only if there exists a $\mu\in[\![\rho^R]\!](\sigma(\mathsf{W}))$ with $\sigma(\mathsf{W})_{\mu(x)}=\sigma(x)$ for all $x\in\mathsf{SVars}(\rho_R)$. As we established before, this holds if and only if $\sigma\models\varphi$ and $(\sigma(x_1),\dots,\sigma(x_k))\in R$. This concludes the "only if"-direction of the proof of the equivalence of selectability by core spanners and by SpLog. As mentioned above, the proof of the "if"-direction proceeds analogously, by using Theorem 21 twice.

As final step, we prove that for every $R\subseteq(\Sigma^*)^k$, the third condition of the claim holds if and only if $R$ is SpLog-selectable. For the "only if"-direction, let $\varphi(\mathsf{W};x_1,\dots,x_k)\in\mathsf{SpLog}(\mathsf{W})$ such that $\sigma\models\varphi$ if and only if $(\sigma(x_1),\dots,\sigma(x_k))\in R$. Now, for any $\psi\in\mathsf{SpLog}(\mathsf{W})$, any $\vec{x}:=(x_1,\dots,x_k)\in(\mathsf{free}(\psi))^k$, define $\psi^R_{\vec{x}}:=(\psi\wedge\varphi)$. Then $\sigma\models\psi^R_{\vec{x}}$ if and only if $\sigma\models\psi$ and $(\sigma(x_1),\dots,\sigma(x_k))\in R$. As $\psi^R_{\vec{x}}$ is a SpLog-formula,we observe that $R$ is SpLog-selectable, which concludes this direction.

For the "if"-direction, assume that $R$ is SpLog-selectable. We define a SpLog-formula $\psi := \bigwedge_{1\leq i\leq k} \exists y_i, z_i \colon (\mathsf{W} = y_i \cdot x_i \cdot z_i)$. Clearly, $\sigma \models \psi$ if and only if $\sigma(x_i) \sqsubseteq \sigma(\mathsf{W})$ for all $1 \leq i \leq k$. As $R$ is SpLog-selectable, there exists a $\varphi \in$ SpLog such that $\sigma \models \varphi$ if and only if $\sigma \models \psi$ and $(\sigma(x_1), \ldots \sigma(x_k)) \in R$. Hence, $\sigma \models (\exists \mathsf{W} \colon \varphi)$ if and only if $(\sigma(x_1), \ldots, \sigma(x_k)) \in R$.

This concludes the proof of this direction, the final claim, and the lemma. ◀

## A.13 Proof of Theorem 28

Before we proceed to the actual proof, we briefly define the semantics for regex, using the ref-word approach by Schmid [25]. First, recall that the syntax of regex can be derived from the syntax of regex formulas, by adding $\mid \&x$ for all $x \in \Xi$ to the recursive definition. We exclude all cases of variable bindings $x\{\alpha\}$ where $\alpha$ contains $\&x$. A *proper regular expression* is a regex that contains no variable bindings and no variable references (i. e., it is just a regular expression).

Instead of defining the language of a regex $\alpha$ directly, we first define its *ref-language* $\mathsf{Ref}(\alpha)$. If $\alpha \in \Sigma \cup \{\varepsilon, \emptyset\}$, $\mathsf{Ref}(\alpha) = \alpha$; and $\mathsf{Ref}(\&x) = x$ for all $x \in \Xi$. Furthermore, $\mathsf{Ref}(\alpha \cdot \beta) = \mathsf{Ref}(\alpha) \cdot \mathsf{Ref}(\beta)$, $\mathsf{Ref}(\alpha \vee \beta) = \mathsf{Ref}(\alpha) \cup \mathsf{Ref}\beta$, and $\mathsf{Ref}(\alpha^*) = \mathsf{Ref}(\alpha)^*$. Finally, $\mathsf{Ref}(x\{\alpha\}) = \vdash_x \cdot \mathsf{Ref}(\alpha) \cdot \dashv_x$.

Intuitively, each subword $\vdash_x w \dashv_x$, where $w$ does not contain $\vdash_x$ or $\dashv_x$, represents that the value $w$ is bound to the variable $x$. Every variable in $r$ that occurs to the right of this subword is now assigned the value $w$, unless another binding changes the value of $x$. More formally, if $ux$ is a prefix of some ref-word $r$, this occurrence of $x$ in $r$ is *undefined* if $u$ does not contain a subword $\vdash_x v \dashv_x$. Otherwise, if $u_1 \vdash_x u_2 \dashv_x u_3 x$ is a prefix of $r$, this occurrence of $x$ *refers to* $u_1 \vdash_x u_2 \dashv_x$ if $u_3$ does not contain $\vdash_x$ (which implies that it also does not contain $\dashv_x$).

The dereference $\mathsf{D}(r)$ of a ref-word $r$ is obtained by first deleting all undefined occurrences of variables. Then, we choose any prefix $u_1 \vdash_x u_2 \dashv_x$ of $r$ for which $u_2 \in \Sigma^*$. We then replace all variables $x$ that refer to this prefix with $u_2$, and rewrite $u_1 \vdash_x u_2 \dashv_x$ to $u_1 u_2$. This process is repeated until we obtain a word from $\Sigma^*$. (See Schmid [25] for more information.)

Finally, we define $\mathcal{L}(\alpha) := \{\mathsf{D}(r) \mid r \in \mathsf{Ref}(\alpha)\}$. Note that for proper regular expressions, $\mathcal{L}(\alpha) = \mathsf{Ref}(\alpha)$.

**Proof.** Let $\alpha$ be a vsf-regex. We first briefly recall a part of the construction that was used in [14] to prove that every language that is generated by a vsf-regex is also a core spanner language (and, hence, a SpLog-language). There, it is first shown that every vsf-regex can be expressed as a finite disjunction of regex paths, where *regex path* is a vsf-regex that is also variable-disjunction free. In other words, a regex path is a vsf-regex $\alpha$ such that for each subexpression $(\alpha_1 \vee \alpha_2)$ of $\alpha$, neither $\alpha_1$ nor $\alpha_2$ contains any variable bindings or references. This is proven by a straightforward rewriting, where for a subexpression $(\alpha_1 \vee \alpha_2)$ that contains variable bindings or references is replaced with $\alpha_1$ and $\alpha_2$, yielding two vsf-regex. This process is repeated until each resulting vsf-regex is also a regex path. For example, $(x\{\mathtt{a}\} \vee x\{\mathtt{b}\})(y\{\mathtt{c}\} \vee y\{\mathtt{d}\})$ is converted into the four regex paths $x\{\mathtt{a}\}y\{\mathtt{c}\}$, $x\{\mathtt{a}\}y\{\mathtt{d}\}$, $x\{\mathtt{b}\}y\{\mathtt{c}\}$, and $x\{\mathtt{b}\}y\{\mathtt{d}\}$. We also refer to this replacement process as *expanding the disjunctions with variables.*

Naturally, this can result in an exponential number of regex paths. As we shall see, SpLog can be used to simulate all these regex paths without explicitly encoding them one by one.

The main problem that the construction has to overcome is handling variables that can be bound multiple times, or not at all. For example, consider the vsf-regex $(x\{\mathtt{a}\} \vee y\{\mathtt{b}\}) \cdot$

$(x\{\mathtt{c}\} \vee y\{\mathtt{d}\}) \cdot \&x \cdot \&y$. There, it is possible to bind each variable once, or one twice and the other not at all, resulting in the words `acc`, `adad`, `bccb`, and `bdd`.

To overcome this, we shall represent each variable $x$ in $\alpha$ with variables $x_0$ to $x_{n(x)}$ in the formula, where $n(x)$ is the highest number of times that $x$ can be bound to a value (hence, in the most recent example, $n(x) = n(y) = 2$). In order to handle these different variables $x_i$, we construct a directed acyclic graph $G(\alpha)$ from $\alpha$ that allows us to see how often the value of each variable $x$ can be assigned, and to which $x_i$ a $\&x$ should refer (further down, we discuss this idea in more details).

First, we represent $\alpha$ as a tree $T(\alpha)$, where the leaves are labeled with proper regular expressions and variable references, while the inner nodes are labeled $\vee$, $\circ$, or $x\{\}$ for some $x \in \Xi$. Hence, if $\alpha$ is a proper regular expression or an $x \in \Xi$, $T(\alpha)$ is simply a node with label $\alpha$. If $\alpha = (\alpha_1 \cdot \alpha_2)$, the root of $T(\alpha)$ is labeled with $\circ$, and it has $T(\alpha_1)$ and $T(\alpha_2)$ as left and right subtree, respectively. Likewise, if $\alpha = (\alpha_1 \vee \alpha_2)$, the root of $T(\alpha)$ is labeled with $\vee$, and $T(\alpha_1)$ and $T(\alpha_2)$ are left and right subtree, respectively. Finally, if $\alpha = x\{\beta\}$, the root of $T(\alpha)$ is labeled with $x\{\}$, and its only subtree is $T(\beta)$. For every node $v$ of $T$, denote its label by $\lambda(v)$, and denote the vsf-Regex from which $v$ was defined by

We now use $T(\alpha)$ to construct a directed acyclic graph $G(\alpha)$. In order to do so, for every node $v$ of $T(\alpha)$, we recursively define the directed acyclic graph $G(v)$ with and a function $\mathsf{snk}(v)$ as follows:

- If $\lambda(v)$ is a proper regular expression or a variable reference, let $G(v) := (V, E)$ with $V := \{v\}$ and $E := \emptyset$, and define $\mathsf{snk}(v) := v$.
- If $\lambda(v) = x\{\}$, let $u$ denote the only child of $v$, and let $(V_u, E_u) := G(u)$. Let $\hat{v}$ be an unlabeled new node, and define $\mathsf{snk}(v) = \hat{v}$. Then $G(v) := (V, E)$, with $V := \{v, \hat{v}\} \cup V_u$ and $E := E_u \cup \{(v, u), (\mathsf{snk}(u), \hat{v})\}$.
- If $\lambda(v) \in \{\circ, \vee\}$, let $u_l$ and $u_r$ denote the left and right child of $v$, respectively. Let $(V_l, E_l) := G(u_l)$ and $(V_r, E_r) := G(u_r)$, and ensure that $(V_l \cap V_r) = \emptyset$. Let $\hat{v}$ be a new node, and define $\mathsf{snk}(v) = \hat{v}$ and $V := \{v, \hat{v}\} \cup V_l \cup V_R$. Furthermore:
  - If $\lambda(v) = \circ$, $E := E_l \cup E_r \cup \{(v, u_l), (\mathsf{snk}(u_l), u_r), (\mathsf{snk}(u_r), \hat{v})\}$.
  - If $\lambda(v) = \vee$, $E := E_l \cup E_r \cup \{(v, u), (\mathsf{snk}(u), \hat{v}) \mid u \in \{u_l, u_r\}\}$.

We use $G(\alpha)$ to denote $G(\mathsf{rt})$, where $\mathsf{rt}$ is the root of $T(\alpha)$.

Now each path in $G(\alpha)$ from $\mathsf{rt}$ to $\mathsf{snk}(\mathsf{rt})$ corresponds to a regex path that can be derived from $\alpha$ when expanding the disjunctions with variables. This is due to the following reasoning: The process of expanding can be understood as passing $T(\alpha)$ top down. If one encounters a disjunction that contains variable bindings or references, one chooses a side of the disjunction, and discards the other. Then obtained regex path corresponds exactly to the path through $G(\alpha)$ that passes from the nodes of the chosen sides to their $\mathsf{snk}$-nodes (over and all other appropriate nodes in between).

An example for $T(\alpha)$ and the construction of $G(\alpha)$ can be found in Figure 4.

For every node $v$ of $G(\alpha)$ and every $x \in \Xi$, we now define $\mathsf{mb}(x, v)$ as the maximal number of nodes with label $x\{\}$ that can appear on a path from $\mathsf{rt}$ to $v$ (not including the label of $v$). Intuitively, $\mathsf{mb}(x, v)$ determines how often a value is assigned to $x$ on the path to $v$. Moreover, if $\lambda(v) = x\{\}$, and $\mathsf{mb}(x, \mathsf{snk}(v)) = i$, we know $x$ has been bound at most $i - 1$ times before this binding, which is why we can represent this binding of $x$ with the variable $x_i$ in the formula. (We also now that there is a path in $G(\alpha)$, and hence a corresponding regex path, where this is exactly the $i$-th binding of $x$.) Recall that by definition, for each subexpression $x\{\beta\}$, $\beta$ cannot contain $x\{\}$ or $\&x$. For an example, see Figure 5.

Furthermore, for each node, $\mathsf{mb}$ can be computed in polynomial time. One way of doing this is using a longest path algorithm (where the edges to nodes with label $x\{\}$ have weight

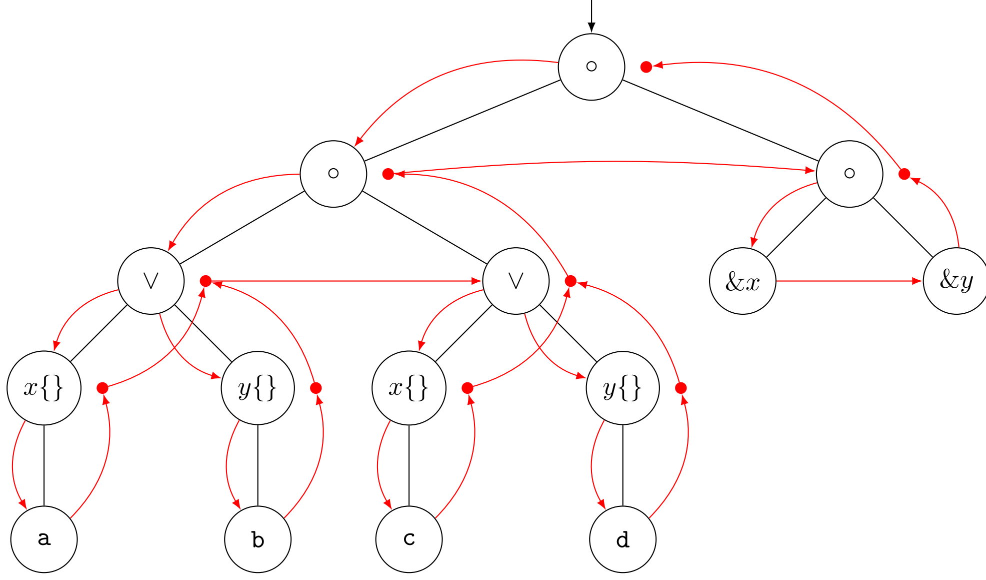



**Figure 4** In black: The tree $T(\alpha)$ for the example $\alpha := ((x\{\mathtt{a}\} \vee y\{\mathtt{b}\}) \cdot (x\{\mathtt{c}\} \vee y\{\mathtt{d}\})) \cdot (\&x \cdot \&y)$ from Section A.13. In red: The edges and the snk-nodes of $G(\alpha)$. Recall that each node of $T(\alpha)$ is also a node of $G(\alpha)$. There are four different paths from the root node to the sink node. Each of these paths corresponds to one of the four regex paths $x\{\mathtt{a}\}x\{\mathtt{c}\}\&x\&y$, $x\{\mathtt{a}\}y\{\mathtt{d}\}\&x\&y$, $y\{\mathtt{b}\}x\{\mathtt{c}\}\&x\&y$, and $y\{\mathtt{b}\}y\{\mathtt{d}\}\&x\&y$ that result from expanding the disjunctions with variables in $\alpha$.

1, and all others have weigth 0), which can be solved in time $O(|V| + |E|)$, cf. Sedgewick and Wayne [26].

The main idea of the construction is that every occurrence of $\&x$ (in some node $v$) is represented by a variable $x_i$ with $i = \mathsf{mb}(x, v)$. To make this work, the formula "fills up" missing variable bindings. More formally, assume that for some $x \in X$, a disjunction has children $u$ and $v$ with $i := \mathsf{mb}(u, x)$ and $j := \mathsf{mb}(v, x)$, such that $i > j$. The formula then extends the subformula for $v$ with assignments $x_{j+1} = x_j$, $x_{j+2} = x_j$ up to $x_i = x_j$.

For every node $v$ of $T(\alpha)$, we define a SpLog-formula $\varphi_v$. Each of these $\varphi_v$ has a characteristic free variable $y_v$ that represents the part of W that is created by the subexpression that is represented by $v$. The formulas are defined as follows:

- If $\lambda(v)$ is a proper regular expression, we define $\varphi_v := (y_v \sqsubseteq \mathsf{W}) \wedge \mathsf{C}_{\lambda(v)}(y_v)$. The expresses that $y_v$ has to be mapped to a word in $\mathcal{L}(\lambda(v))$, and needs no further explanation.
- If $\lambda(v) = \&x$, let $\varphi_v := (y_v = x_{\mathsf{mb}(x,v)})$. This expresses that $y_v$ has to be mapped to the same value as $x_i$, which is supposed to contain the most recent binding of $x$ (this shall be ensured by the formulas for conjunctions further down).
- If $\lambda(v) = x\{\}$, let $u$ denote the child of $v$ in $T(\alpha)$, and define

$$\varphi_v := \exists y_u\colon (y_v = y_u) \wedge (x_{\mathsf{mb}(x,\mathsf{snk}(v))} = y_u).$$

  As explained above, if $i := \mathsf{mb}(x, \mathsf{snk}(v))$, then $x$ was bound at most $i - 1$ times before the most recent binding. Hence, we store the most current value of $x$ in $x_i$. The task of generating the word that is represented by $y_v$ is then delegated to $y_u$.
- If $\lambda(v) = \circ$, let $u_l$ and $u_r$ to denote the left and the right child of $v$ in $T(\alpha)$. We define

$$\varphi_v := \exists y_{u_l}, y_{u_r}\colon \big((y_v = y_{u_l} \cdot y_{u_r}) \wedge \varphi_{u_l} \wedge \varphi_{u_r}\big).$$

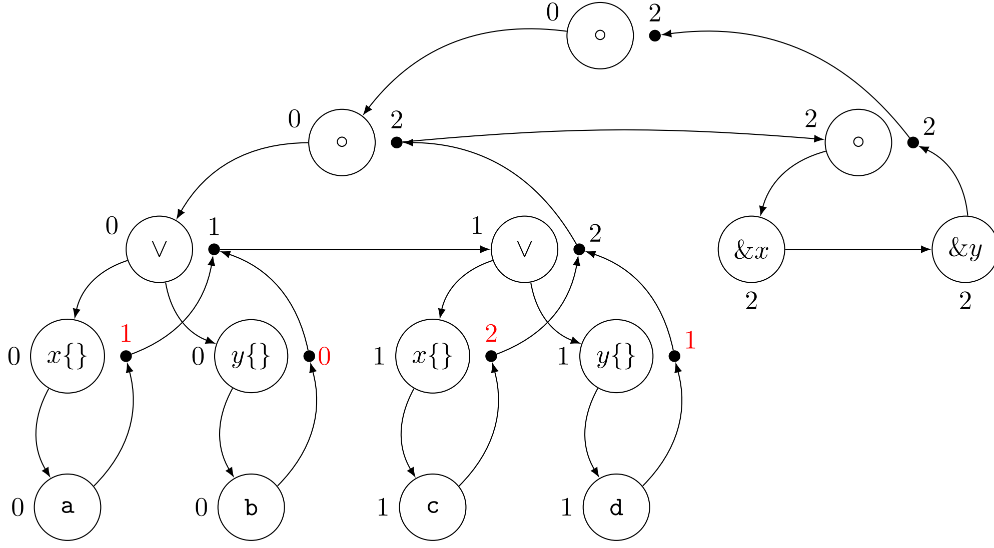



**Figure 5** The graph $G(\alpha)$ for the example from Figure 5. The numbers indicate the value $\mathsf{mb}(x, v)$, where $v$ is the respective node. In order to use this example for the values for $\mathsf{mb}(y, v)$, the numbers that are marked in red have to be changed: From left to right, 1, 0, 2, 1 are replaced with 0, 1, 1, 2, respectively. For all other nodes, $\mathsf{mb}(x, v) = \mathsf{mb}(y, v)$ holds.

This is also straightforward: $y_v$ is a concatenation of $y_{u_l}$ and $y_{u_r}$, and these words are handled by the respective subformulas.

- If $\lambda(v) = \vee$, use $u_1, u_2$ to denote the children of $v$ in $T(\alpha)$, without any special regard to which is left or right. We define

$$\varphi_v := \Big(\exists y_{u_1} \colon (y_v = y_{u_1}) \wedge \varphi_{u_1} \wedge \bigwedge_{x_i \in X} (x_i \sqsubseteq W) \wedge \bigwedge_{\substack{x \in \mathsf{var}(\alpha), \\ m_1^x < i \leq m_2^x}} (x_i = x_{m_1^x})\Big)$$
$$\vee \Big(\exists y_{u_2} \colon (y_v = y_{u_2}) \wedge \varphi_{u_2} \wedge \bigwedge_{x_i \in X} (x_i \sqsubseteq W) \wedge \bigwedge_{\substack{x \in \mathsf{var}(\alpha), \\ m_2^x < i \leq m_1^x}} (x_i = x_{m_2^x})\Big)$$

where $X := \{x_i \mid x \in \mathsf{var}(\alpha), 0 \leq i \leq \mathsf{mb}(x, \mathsf{snk}(\mathsf{rt}))\}$, and $m_l^x := \mathsf{mb}(x, \mathsf{snk}(u_l))$ for $l \in \{1, 2\}$. This formula consists of two almost identical subformulas, which we now examine from left to right: First, the subformula states that $y_v$ is determined by $y_{u_l}$, and delegates the task of determining $y_{u_l}$ to $\varphi_{u_l}$. Next, the conjunction $\bigwedge_{x_i \in X}(x_i \sqsubseteq W)$ ensures that the formula is safe (at the end of this section, we discuss how this coild be optimized). Finally, the last part of the formula realizes the aforementioned “filling up”. Assume that $l = 1$ and $i < j$, where $i := m_1^x$ and $j := m_2^x$ for some $x$. Then $\varphi_{u_1}$ defines $x_{i+1} = x_i$, $x_{i+2} = x_i$ up to $x_j = x_i$.

We then combine these subformulas into

$$\varphi := \exists y_{\mathsf{rt}}, \vec{x} \colon \big((\mathsf{W} = y_{\mathsf{rt}}) \wedge \varphi_{\mathsf{rt}} \wedge \bigwedge_{x \in \mathsf{var}(\alpha)} (x_0 = \varepsilon)\big),$$

where $\vec{x}$ is any ordering of $\{x_0 \mid x \in \mathsf{var}(\alpha)\}$. As mentioned above, $\mathsf{mb}$ can be computed in time that is polynomial in the size of $\alpha$; hence, $\varphi$ can be constructed in polynomial time. All we have to do is to verify $\mathcal{L}(\varphi) = \mathcal{L}(\alpha)$.

For both directions of this claim, we observe the following invariant: If $v$ is a node of $T(\alpha)$ with $\lambda(v) = \vee$, and $i := \mathsf{mb}(x, v)$ and $j := \mathsf{mb}(x, \mathsf{snk}(v))$, then $\varphi_v$ assigns exactly the variables $x_l$ with $i < l \leq j$. Each of these assignments can happen either through a $x_l = y_u$ (due to a variable binding $x\{\}$), or due to some $x_l = x_{\hat{l}}$ with $i \leq \hat{l} < l$ (from the disjunction at $v$, or from a disjunction in a subexpression of that disjunction).

Now, for each $w \in \mathcal{L}(\alpha)$, there is a regex path $\hat{\alpha}$ that is obtained from $\alpha$ by expanding the disjunctions with variables, and $w \in \mathcal{L}(\hat{\alpha})$. As mentioned above, $\hat{\alpha}$ corresponds to a path in $G(\alpha)$ from $\mathsf{rt}$ to $\mathsf{snk}(\mathsf{rt})$, which is equivalent to a choice of disjunctions in $\varphi$. For every variable reference $\&x$ in $\hat{\alpha}$, the corresponding formula uses a variable $x_i$, where $x_i = x_j$ holds, and $j$ is the number of the most recent binding for $x$ (in particular, if $x$ has never been bound, it defaults to $x_0 = \varepsilon$). Hence, $w \in \mathcal{L}(\varphi)$ holds. Likewise, if $w \in \mathcal{L}(\varphi)$, we can follow the corresponding $\sigma \models \varphi$ with $\sigma(\mathsf{W}) = w$ along the structure of $\varphi$, updating the substitution whenever we encounter an existential quantifier. Whenever we encounter a disjunction, there is at least one side where the current substitution is satisfied. This side corresponds to a node in $T(\alpha)$, and we can use this to construct an according path in $G(\alpha)$. ◀

As mentioned above, this construction is not yet optimal, as we include $\bigwedge_{x_i \in X}(x_i \sqsubseteq W)$ in every side of a disjunction. By analyzing the respective subexpressions, we could determine which variables $x_i$ are bound or referenced in the other side of the disjunction, and only include those that are needed to make the formula safe.

Another potential use of $\mathsf{mb}$ is the simplification of $\alpha$. Mention optimizations: If $\lambda(v) = \&x$ and $\mathsf{mb}(x, v) = 0$, we know that this occurrence of $\&x$ can never reference a non-default value, and we can safely replace it with $\varepsilon$. Likewise, we could use this to identify variable bindings that are never referenced, and remove these from the expression.

## A.14 Proof of Lemma 30

**Proof.** First, we ensure that for every subformula of $\varphi$ that has the form $\exists x\colon \psi$, $x$ does not appear in $\varphi$ outside of $\psi$. In particular, this means that quantifiers do not rebind variables, and no two quantifiers range over the same variable. This is easily achieved by renaming any pair of variables that violates this criterion.

The DPC-normal form can then be computed by applying the following rewriting rules:

$$((\varphi_1 \vee \varphi_2) \wedge \varphi_C) \to (\varphi_1 \wedge \varphi_C) \vee (\varphi_2 \wedge \varphi_C), \qquad (R_1)$$
$$((\exists x\colon \varphi_1) \wedge \varphi_C) \to (\exists x\colon (\varphi_1 \wedge \varphi_C)), \qquad (R_2)$$
$$(\exists x\colon (\varphi_1 \vee \varphi_2)) \to ((\exists x\colon \varphi_1) \vee (\exists x\colon \varphi_2)), \qquad (R_3)$$

where $x \in \Xi$, $\varphi_1, \varphi_2 \in \mathsf{SpLog}$, and $\varphi_C$ is a $\mathsf{SpLog}$-formula or a constraint. These rules are also applied modulo commutation of $\wedge$ and $\vee$; i.e., $\varphi_C \wedge (\varphi_1 \vee \varphi_2)$ is rewritten to $(\varphi_1 \wedge \varphi_C) \vee (\varphi_2 \wedge \varphi_C)$.

Intuitively, the rules can be understood as follows: If one views the syntax tree of the formula, $R_1$ moves $\vee$ over $\wedge$, $R_2$ moves $\exists$ over $\wedge$, and $R_3$ moves $\vee$ over $\exists$. Hence, when no more rules can be applied, the resulting formula has $\vee$ over $\exists$, and $\exists$ over $\wedge$, which is exactly the order that is required by DPC-normal form.

Furthermore, note that the rules preserve the syntactic requirements of $\mathsf{SpLog}$-formulas. In particular, as the equations are not rewritten and no new existential quantifiers are introduced, it suffices to check that the resulting formulas are safe. For example, consider $R_1$. For every $\varphi \in \mathsf{SpLog}$ with $\varphi = ((\varphi_1 \vee \varphi_2) \wedge \varphi_C)$, $\mathsf{free}(\varphi_1) = \mathsf{free}(\varphi_2)$ must hold. This has two consequences. Firstly, $\mathsf{free}(\varphi_1 \wedge \varphi_C) = \mathsf{free}(\varphi_2 \wedge \varphi_C)$, which means that the disjunction that

results from $R_1$ is safe. Secondly, if $\varphi_C$ is some constraint $C_A(x)$, $x \in \mathsf{free}(\varphi_1 \vee \varphi_2)$ must hold. Hence, as $\mathsf{free}(\varphi_1) = \mathsf{free}(\varphi_2) = \mathsf{free}(\varphi_1 \vee \varphi_2)$, the resulting subformulas $(\varphi_1 \wedge \varphi_C)$ and $(\varphi_2 \wedge \varphi_C)$ are safe. ◀

## A.15 Proof of Lemma 31

**Proof.** Let $\varphi(\mathsf{W}) \in \mathsf{SpLog}$, and let $a \in \Sigma$. Without loss of generality, we can assume that $\varphi$ is a prenex conjunction. This holds for the following reason: According to Lemma 30, we can assume that $\varphi$ is in DPC-normal form, with $\varphi := \bigvee \varphi_i$, and $\mathcal{L}(\varphi) = \bigcup \mathcal{L}(\varphi_i)$. Hence, $\mathcal{L}(\varphi)\,/\mathrm{a} = \bigcup(\mathcal{L}(\varphi_i)\,/\mathrm{a})$. Hence, let

$$\varphi := \exists x_1, \ldots, x_k \colon (\bigwedge_{i=1}^{m} \eta_i \wedge \bigwedge_{j=1}^{n} C_j)$$

with $k, n \geq 0$ and $m \geq 1$, and $\eta_i = (\mathsf{W}, \alpha_i)$ for some $\alpha \in (X \cup \Sigma)^*$, where $X := \{x_1, \ldots, x_k\}$.

Our goal is to the $\alpha_i$ into a form where we can easily split off $a$ at the right side. Hence, we consider all possibilities which variables generates the rightmost letter in a word $w \in \mathcal{L}(\varphi)$. (As some variables might be erased, there can be multiple possibilities). To this purpose, for each set $N \subseteq X$, we define a morphism $\pi_N \colon (X \cup \Sigma)^* \to (N \cup \Sigma)^*$ by $\pi_N(c) := c$ for all $c \in \Sigma$, $\pi_N(x) := x$ for $x \in N$, and $\pi_N(x) := \varepsilon$ for $x \in (X - N)$. In other words, variables in $N$ are not erased, while all other variables are erased. For each of these $N$, we now define a formula

$$\varphi_N := \exists \vec{x} \colon (\bigwedge_{i=1}^{m} (\mathsf{W} = \pi_N(\alpha_i)) \wedge \bigwedge_{j=1}^{n} C_j \wedge \bigwedge_{x \in N} (x \neq \varepsilon) \wedge \bigwedge_{x \in (X-N)} (x = \varepsilon)),$$

where $\vec{x} = (x_1, \ldots, x_k)$. Some (or all) of these formulas might not be satisfiable (e. g., when $x = \varepsilon$ is forbidden by a constraint on $x$), but this is not a problem.

Up to now, we have only rewritten parts of $\varphi$ without changing its language, and we observe that $\varphi \equiv \bigvee_{\emptyset \subset N \subseteq X} \varphi_N$ (unless $\varepsilon \in \mathcal{L}(\varphi)$, which is expressed by no $\varphi_N$). Our next goal is to focus on $\mathcal{L}(\varphi) \cap (\Sigma^* \cdot a)$. In particular, for every $\varphi_N$, we want to construct a formula $\psi_N$ with $\mathcal{L}(\psi_N) = \mathcal{L}(\varphi_N) \cap (\Sigma^* \cdot a)$.

In this sublanguage, every variable that is the rightmost symbol of some $\pi_N(\alpha_i)$ has to be substituted with a word that ends on $a$. In order to simulate this, we first define the set of these variables as

$$R_N := \{x \in N \mid \text{some } \pi_N(\alpha_i) \text{ ends on } x\}.$$

We use this to define a morphism $s_N \colon (N \cup \Sigma)^* \to (N \cup \Sigma^*)$ by $s_N(c) := c$ for all $c \in \Sigma$, $s_N(x) := x$ for all $x \in (N - R_N)$, and $s_N(x) := x \cdot a$ for all $x \in R_N$. We use this to define $\beta_{N,i} := s_N(\pi_N(\alpha_i))$ for $1 \leq i \leq m$. In order to construct an appropriate formula, we need to modify some of the constraints. For each $1 \leq j \leq n$, there exist an NFA $A$ and a $x \in X$ such that $C_j = \mathsf{C}_A(x)$. If $x \notin R_N$, we define $C_j^q := C_j$. On the other hand, if $x \in R_N$, let $C_j^q := \mathsf{C}_{A_q}(x)$, where $A_q$ is an NFA with $\mathcal{L}(A_q) = \mathcal{L}(A)\,/\mathrm{a}$. As the class of regular languages is closed under $/\mathrm{a}$, such an $A_q$ always exists (and although $\mathcal{L}(A_q) = \emptyset$ might hold, this simply results in a formula that is not satisfiable). We combine these definitions to

$$\psi_N := \exists \vec{x} \colon (\bigwedge_{i=1}^{m} (\mathsf{W} = \beta_{N,i}) \wedge \bigwedge_{j=1}^{n} C_j^q \wedge \bigwedge_{x \in (N - R_N)} (x \neq \varepsilon) \wedge \bigwedge_{x \in (X-N)} (x = \varepsilon))$$

As we replaced each $x \in R_n$ with $x \cdot a$, we can (and have to) exclude these variables from the conjunction that requires $x \neq \varepsilon$. Due to our definitions of the $\beta_{N,i}$ and $C_j^q$, $\mathcal{L}(\psi_N) = \mathcal{L}(\varphi_N) \cap (\Sigma^* \cdot a)$ holds as intended.

Now we need for final step, splitting of the $a$. Our goal is to define formulas $\psi_N^q$ with $\mathcal{L}(\psi_N^q) = \mathcal{L}(\psi_N)\,/\mathrm{a}$. We distinguish two cases: Firstly, if, for some $N$, any of the $\beta_{N,i}$ ends on some terminal from $\Sigma - \{a\}$, we know that $\mathcal{L}(\varphi_N) \cap (\Sigma^* \cdot a) = \emptyset$, which is equivalent to $\mathcal{L}(\varphi_N)\,/\mathrm{a} = \emptyset$. Hence, we can discard this choice of $N$. To simplify the presentation, we then define $\psi_N^q$ to be some formula that is not satisfiable, like $(\mathsf{W} = a) \wedge (\mathsf{W} = aa)$.

Otherwise, we know that for this $N$, each $\beta_{N,i}$ has to end on $a$ (no $\beta_{N,i}$ can end on a variable, as $s_N$ replaces each variable $x$ that occurs at the right of some $\beta_i$ with $x \cdot a$). Therefore, for each $1 \leq i \leq m$, there exists a well-defined $\gamma_{N,i}$ with $\gamma_{N,i} = \beta_{N,i} \cdot a$. Hence, we can define

$$\psi_N^q := \exists \vec{x}\colon \big(\bigwedge_{i=1}^{m}(\mathsf{W} = \gamma_{N,i}) \wedge \bigwedge_{j=1}^{n} C_j^q \wedge \bigwedge_{x\in(N-R_N)} (x \neq \varepsilon) \wedge \bigwedge_{x\in(X-N)} (x = \varepsilon)\big),$$

and we observe that $\mathcal{L}(\psi_N^q) = \mathcal{L}(\psi_N)\,/\mathrm{a}$. All that remains is to combine the formulas into a single formula $\psi := \bigvee_{\emptyset\subset N\subseteq X} \psi_N^q$. As $\mathsf{free}(\psi_N^q) = \{\mathsf{W}\}$ for each $N$, this is indeed a $\mathsf{SpLog}$-formula, and by our previous observations, we can state

$$\begin{aligned}\mathcal{L}(\psi) = \bigcup_{\emptyset\subset N\subseteq X} \mathcal{L}(\psi_N^q) &= \bigcup_{\emptyset\subset N\subseteq X} \mathcal{L}(\psi_N)\,/\mathrm{a}\\ &= \bigcup_{\emptyset\subset N\subseteq X} (\mathcal{L}(\varphi_N) \cap (\Sigma^* \cdot a))\,/\mathrm{a}\\ &= \bigcup_{\emptyset\subset N\subseteq X} \mathcal{L}(\varphi_N)\,/\mathrm{a}\\ &= \big(\bigcup_{\emptyset\subset N\subseteq X} \mathcal{L}(\varphi_N)\big)\,/\mathrm{a} = \mathcal{L}(\varphi)\,/\mathrm{a}\,.\end{aligned}$$

Hence, for every $\mathsf{SpLog}$-language $L$ and every $a \in \Sigma$, $L\,/\mathrm{a}$ is also a $\mathsf{SpLog}$-language. ◀

## A.16 Proof of Proposition 32

**Proof.** We first show the equivalence of statements 1 and 2, and then of statements 1 and 3.

**Equivalence of statements 1 and 2:** If $\mathcal{L}(\mathsf{EC}^{\mathsf{reg}}) = \mathcal{L}(\mathsf{SpLog})$, the question is trivially decidable. Hence, assume that $\mathcal{L}(\mathsf{EC}^{\mathsf{reg}}) \neq \mathcal{L}(\mathsf{SpLog})$. We use the following version of Greibach's Theorem (Theorem 8.14 in [17]):

▶ Greibach's Theorem. Let $\mathcal{C}$ be a class of languages that is effectively closed under concatenation with regular sets and union, and for which $= \Sigma^*$ is undecidable for any sufficiently large fixed $\Sigma$. Let $P$ be any non-trivial property that is true for all regular languages and that is preserved under $/\mathrm{a}$, where $a \in \Sigma$. Then $P$ is undecidable for $\mathcal{C}$.

We choose $\mathcal{C}$ as the class of $\mathsf{EC}^{\mathsf{reg}}$-languages, and $P$ as the property that $L$ is a $\mathsf{SpLog}$-language. In order to apply Greibach's Theorem, we establish the following claims:

1. From every $\varphi \in \mathsf{EC}^{\mathsf{reg}}$ and every NFA $A$, we can compute $\varphi_L, \varphi_R \in \mathsf{EC}^{\mathsf{reg}}$ with $\mathcal{L}(\varphi_L) = \mathcal{L}(A) \cdot \mathcal{L}(\varphi)$ and $\mathcal{L}(\varphi_R) = \mathcal{L}(\varphi) \cdot \mathcal{L}(A)$.
   *Proof:* Given $\varphi \in \mathsf{EC}^{\mathsf{reg}}$ and an NFA $A$, we define:

   $$\begin{aligned}\varphi_L(w) &:= \exists u, v\colon (w = u \cdot v) \wedge L_A(v) \wedge \varphi(v),\\ \varphi_R(w) &:= \exists u, v\colon (w = u \cdot v) \wedge \varphi(u) \wedge L_A(v).\end{aligned}$$

   Then $\mathcal{L}(\varphi_L) = \mathcal{L}(A) \cdot \mathcal{L}(\varphi)$ and $\mathcal{L}(\varphi_R) = \mathcal{L}(\varphi) \cdot \mathcal{L}(A)$ hold.

2. From all $\varphi_1, \varphi_2 \in \mathsf{EC}^{\mathsf{reg}}$, we can compute $\varphi \in \mathsf{EC}^{\mathsf{reg}}$ with $\mathcal{L}(\varphi) = \mathcal{L}(\varphi_1) \cup \mathcal{L}(\varphi_2)$.
   *Proof:* Given $\varphi_1, \varphi_2 \in \mathsf{EC}^{\mathsf{reg}}$, we define $\varphi(w) := (\varphi_1(w) \vee \varphi_2(w))$, and observe $\mathcal{L}(\varphi) = \mathcal{L}(\varphi_1) \cup \mathcal{L}(\varphi_2)$.
3. For $\Sigma$ with $|\Sigma| \geq 2$, $\mathcal{L}(\varphi) = \Sigma^*$ is undecidable when given a $\varphi \in \mathsf{SpLog}$.
   *Proof:* This follows from Theorem 28, together with the fact that this problem is undecidable for vsf-Regex if $|\Sigma| \geq 2$. (See [13], or [14]).
4. $P$ is a non-trivial property.
   *Proof:* In other words, we need to show that $\emptyset \subset P \subset \mathcal{C}$. The first proper inclusion follows from $\mathcal{L}(\mathsf{SpLog}) \neq \emptyset$ and $\mathcal{L}(\mathsf{SpLog}) \subseteq \mathcal{L}(\mathsf{EC}^{\mathsf{reg}})$. The second proper inclusion is our initial assumption.
5. $P$ is true for all regular languages.
   *Proof:* Every regular language is a $\mathsf{SpLog}$-language by definition.
6. $P$ is preserved under $/a$.
   *Proof:* Lemma 31.

Hence, Greibach's Theorem applies, under the assumption that $\mathcal{L}(\mathsf{EC}^{\mathsf{reg}}) \neq \mathcal{L}(\mathsf{SpLog})$.

**Equivalence of statements 1 and 3:** In other words, we prove that $\mathcal{L}(\mathsf{EC}^{\mathsf{reg}}) = \mathcal{L}(\mathsf{SpLog})$ if and only if $\mathcal{L}(\mathsf{SpLog})$ is closed under the prefix operator.

For the only if direction, assume $\mathcal{L}(\mathsf{EC}^{\mathsf{reg}}) = \mathcal{L}(\mathsf{SpLog})$, and choose any $\mathcal{L}(\varphi) \in \mathcal{L}(\mathsf{SpLog})$. We then define $\psi := \exists y, z\colon ((y = xz) \wedge \varphi(y))$. Then $\mathcal{L}(\psi) = \{x \mid x \text{ is prefix of some } y \in \mathcal{L}(\varphi)\}$. This shows that the language of all prefixes of words from $\mathcal{L}(\varphi)$ is an $\mathsf{EC}$-language, and as we assumed $\mathcal{L}(\mathsf{EC}^{\mathsf{reg}}) = \mathcal{L}(\mathsf{SpLog})$, it is also a $\mathsf{SpLog}$-language, which proves the direction of the claim.

For the if direction, assume that $\mathcal{L}(\mathsf{SpLog})$ is closed under the prefix operator, and choose any $\mathcal{L}(\varphi) \in \mathcal{L}(\mathsf{EC}^{\mathsf{reg}})$. Assume that $\mathsf{free}(\varphi) = \{x\}$. As explained by Diekert [7], $\varphi$ can be converted into an equivalent word equation $\eta = (\eta_L, \eta_R)$, which yields an $\mathsf{EC}^{\mathsf{reg}}$-formula $\varphi_E \equiv \varphi$, that is of the form $\varphi_E = \exists \vec{y}\colon (\eta \wedge C)$, where $\vec{y}$ is a sequence of variables, and $C$ is a conjunction of constraints. As $\mathsf{free}(\varphi_E) = \mathsf{free}(\varphi) = \{x\}$, $x$ does not occur in $\vec{y}$.

Now, \$ be a new terminal letter, and let $\mathsf{W}$ be a new variable that does not occur in $\varphi_E$, and define $\psi := \exists \vec{y}\colon ((\mathsf{W} = x\$\eta_L) \wedge (\mathsf{W} = x\$\eta_R) \wedge C)$. Then $\psi$ is a $\mathsf{SpLog}(\mathsf{W})$-formula, and $\mathcal{L}(\psi) = \{\sigma(x)\$\sigma(\eta_L) \mid \sigma \models \varphi_E\}$. Now, let

$$L_1 := \{u \mid \text{there is a } v \in (\Sigma \cup \{\$\})^* \text{ with } uv \in \mathcal{L}(\psi)\},$$
$$L_2 := L_1 \cap (\Sigma^* \cdot \$),$$
$$L_3 := L_2 \,/\$\,.$$

As $\mathcal{L}(\psi) \in \mathcal{L}(\mathsf{SpLog})$, $L_1$ is a $\mathsf{SpLog}$-language, due to our initial assumption. As $\mathsf{SpLog}$-languages are closed under intersection with regular languages (see the previous equivalence, claim 1), $L_2$ is a $\mathsf{SpLog}$-language, and so is $L_3$ (due to Lemma 31). We now observe $L_3 = \{\sigma(x) \mid \sigma \models \varphi_E\} = \mathcal{L}(\varphi_E) = \mathcal{L}(\varphi)$. Hence, $\mathcal{L}(\varphi) \in \mathcal{L}(\mathsf{SpLog})$.

◀

## A.17 Proof of Lemma 33

**Proof.** We base our proof on Theorem 1.1 from [16], which states that the class of bounded regular languages is exactly the smallest class that contains all finite languages, all languages $w^*$ with $w \in \Sigma^*$, and is closed under finite union and concatenation.

As the set of $\mathsf{EC}$-languages is closed under finite union by definition, every finite language is an $\mathsf{EC}$-language. Closure under concatenation is also straightforward. Finally, as shown in

Theorem 5 in [20], for every $w \in \Sigma^*$, $w^*$ is an EC-language. Hence, the class of EC-languages contains the class of bounded regular languages. ◀

## A.18 Proof of Theorem 34

**Proof.** Let $\varphi \in \mathsf{SpLog}(\mathsf{W})$ such that $\mathcal{L}(\varphi)$ is bounded. Hence, $\mathcal{L}(\varphi) \subseteq B$ for some $B := w_1^* \cdots w_k^*$, with $k \geq 1$ and $w_1, \ldots, w_k \in \Sigma^*$.

Our goal is to show that all languages in constraints in $\varphi$ can be replaced with bounded regular languages. By Lemma 33, these constraints can then be replaced with EC-formulas, which allows us to rewrite $\varphi$ into an equivalent EC-formula.

To this purpose, consider any constraint $\mathsf{C}_A(x)$ in $\varphi$, together with a substitution $\sigma$ that is obtained from a substitution $\hat{\sigma} \models \varphi$. As $\varphi$ may contain existential quantifiers, we do not consider $\hat{\sigma}$ directly, but we observe that $\sigma(\mathsf{W}) = \hat{\sigma}(\mathsf{W})$ must hold. Furthermore, $\sigma(\mathsf{W}) \in B$, as $\mathcal{L}(\varphi) \subseteq B$.

As $\varphi$ is a $\mathsf{SpLog}(\mathsf{W})$-formula, $\sigma(x) \sqsubseteq \sigma(\mathsf{W})$, which implies $\sigma(x) \in B_{\sqsubseteq}$, where $B_{\sqsubseteq} := \{u \mid u \sqsubseteq v \text{ for some } v \in B\}$. As $B$ is a regular language; and as the class of regular languages is closed under considering all subwords, $B_{\sqsubseteq}$ is regular as well. The class of regular languages is also closed under intersection; thus, there exists an NFA $A_{\sqsubseteq}$ with $\mathcal{L}(A_{\sqsubseteq}) = \mathcal{L}(A) \cap B_{\sqsubseteq}$. Therefore, we can replace the constraint $\mathsf{C}_A(x)$ in $\varphi$ with $\mathsf{C}_{A_{\sqsubseteq}}(x)$, without changing the languages $\mathcal{L}(\varphi)$.

All that remains to be shown is that each $\mathcal{L}(A_{\sqsubseteq})$ is bounded: Then, by Lemma 33, there exists an $\varphi_{A_{\sqsubseteq}} \in \mathsf{EC}$ with $\mathcal{L}(\varphi_{A_{\sqsubseteq}}) = \mathcal{L}(A_{\sqsubseteq})$, and we can replace $\mathsf{C}_{A_{\sqsubseteq}}(x)$ with $\varphi_{A_{\sqsubseteq}}$.

We begin by showing that $B_{\sqsubseteq}$ is bounded regular. For every $w \in \Sigma^*$, let $\mathsf{pref}(w) := \{u \in \Sigma^* \mid w = uv\}$ and $\mathsf{pref}(w) := \{v \in \Sigma^* \mid w = uv\}$. Now define the language

$$B_T := \bigcup_{i=1}^{k} \bigcup_{j=i}^{k} \bigcup_{p \in \mathsf{pref}(w_i)} \bigcup_{s \in \mathsf{suff}(w_j)} p \cdot w_i^* \cdots w_j^* \cdot s,$$

and observe that $B_{\sqsubseteq} \subseteq B_T$. This inclusion holds as, for every $u \in B_{\sqsubseteq}$, there is a $v \in B$ with $u \sqsubseteq v$.

Note that $B_T$ is a finite union of languages $L_{p,i,j,s} = p \cdot w_i^* \cdots w_j^* \cdot s$, as $1 \leq i \leq j \leq k$, and for each $w \in \Sigma^*$, $|\mathsf{pref}(w)| = |\mathsf{suff}(w)| \leq |w| + 1$. Furthermore, each $L_{p,i,j,s}$ is a bounded regular language, and as the set of bounded regular languages is closed under finite union (cf. [16]), $B_T$ is bounded regular as well. As every subset of a bounded language is bounded, $\mathcal{L}(A_{\sqsubseteq}) \subseteq B_{\sqsubseteq} \subseteq B_T$ allows us to conclude that $\mathcal{L}(A_{\sqsubseteq})$ is bounded regular.

Hence, each $\mathsf{C}_{A_{\sqsubseteq}}(x)$ can be replaced with an equivalent $\varphi_{A_{\sqsubseteq}} \in \mathsf{EC}$, which means that $\varphi$ can be rewritten into an equivalent EC-formula. ◀

## A.19 Proof of Proposition 37

**Proof.** The proof follows the same outline as the example that was given for the equal length relation: We first define three languages $L_1$ to $L_3$, each of which is shown not to be a SpLog-language. For each of the relations, we then show that SpLog-selectability of this relation would imply $L_i \in \mathcal{L}(\mathsf{SpLog})$ for some $i$. Assume $\mathtt{a}, \mathtt{b}, \in \Sigma$. We define the following languages:

$L_1 := \{\mathtt{a}^i \mathtt{b}^j \mid 0 \leq i \leq j\}$,
$L_2 := \{\mathtt{a}^i (\mathtt{ba})^j \mid 0 \leq i \leq j\}$,
$L_3 := \{(\mathtt{abaabb})^i (\mathtt{bbaaba})^i \mid i \geq 0\}$.

Note that these languages ar all bounded. Hence, according to Theorem 34, it suffices to show that they are not EC-languages, which we do by applying Theorem 36.

*ad* $L_1$*:* This proof is almost identical to the example: Assume that $L_1$ is an EC-language, and choose $Q_1 := \mathtt{b}$. Then there exists a $k_1$ that satisfies Theorem 36. Let $w_1 := \mathtt{a}^{k_1+2}\mathtt{b}^{k_1+2}$, and observe the $\mathscr{F}_Q$-factorization $w_1 = \mathtt{a}^{k+2}\mathtt{b} \cdot \mathtt{b}^{k+1} \cdot \varepsilon$. Hence, $\exp_{Q_1}(w_1) = k+1 > k$, and there exists an $u = \mathtt{a}^{k_1+2}\mathtt{b}^{k_1+2-j}$ with $j > 0$ and $u \in L_1$. As $k_1 + 2 > k_1 + 2 - j$, we observe the contradiction $u_1 \notin L_1$. Therefore, $L_1 \notin \mathcal{L}(\mathsf{EC})$, and $L_1 \notin \mathcal{L}(\mathsf{SpLog})$.

*ad* $L_2$*:* The proof proceeds as for $L_1$, by choosing $Q_2 := \mathtt{ba}$, $w_2 = \mathtt{a}^{k+2}(\mathtt{ba})^{k+2}$, and observing the $\mathscr{F}_Q$-factorization $w_2 = \mathtt{a}^{k+2}\mathtt{ba} \cdot (\mathtt{ba})^{k+1} \cdot \varepsilon$.

*ad* $L_3$*:* Assume that $L_3$ is an EC-language, and choose $Q_3 := \mathtt{abaabb}$. Let $k_3$ be the constant from Theorem 36, and choose $w_3 := (\mathtt{abaabb})^{k_3+2}(\mathtt{bbaaba})^{k_3+2}$. The $\mathscr{F}_Q$-factorization is $w_3 = \varepsilon \cdot (\mathtt{abaabb})^{k_3+1} \cdot \mathtt{abaabb}(\mathtt{bbaaba})^{k_3+2}$; hence, $\exp_{Q_3}(w_3)) = k_3 + 1$. According to Theorem 36, there is a $u_3 = (\mathtt{abaabb})^{k_3+2-j}(\mathtt{bbaaba})^{k_3+2}$ with $u_3 \in L$, $j > 0$, and we obtain a contradiction as in the previous cases.

Now assume that some $R \in \{R_{\text{scatt}}, R_{\text{num}(a)}, R_{\text{permut}}, R_{\text{rev}}, R_{<}\}$ is SpLog-selectable, with some corresponding $\varphi_R(\mathsf{W}; x, y) \in \mathsf{SpLog}$. We construct the following formulas:

$$\varphi_1(\mathsf{W}) := \exists x, y\colon (\mathsf{W} = x \cdot y) \wedge \mathsf{C}_{\mathtt{a}^*}(x) \wedge \mathsf{C}_{(\mathtt{ba})^*}(y) \wedge \varphi_{R_{\text{scatt}}}(\mathsf{W}; x, y),$$
$$\varphi_2(\mathsf{W}) := \exists x, y\colon (\mathsf{W} = x \cdot y) \wedge \mathsf{C}_{(\mathtt{abaabb})^*}(x) \wedge \mathsf{C}_{(\mathtt{bbaaba})^*}(y) \wedge \varphi_{R_{\text{num}(\mathtt{a})}}(\mathsf{W}; x, y),$$
$$\varphi_3(\mathsf{W}) := \exists x, y\colon (\mathsf{W} = x \cdot y) \wedge \mathsf{C}_{(\mathtt{abaabb})^*}(x) \wedge \mathsf{C}_{(\mathtt{bbaaba})^*}(y) \wedge \varphi_{R_{\text{permut}}}(\mathsf{W}; x, y),$$
$$\varphi_4(\mathsf{W}) := \exists x, y\colon (\mathsf{W} = x \cdot y) \wedge \mathsf{C}_{(\mathtt{abaabb})^*}(x) \wedge \mathsf{C}_{(\mathtt{bbaaba})^*}(y) \wedge \varphi_{R_{\text{rev}}}(\mathsf{W}; x, y),$$
$$\varphi_5(\mathsf{W}) := \exists x, y\colon (\mathsf{W} = x \cdot y) \wedge \mathsf{C}_{\mathtt{a}^*}(x) \wedge \mathsf{C}_{\mathtt{b}^*}(y) \wedge \varphi_{R_{<}}(\mathsf{W}; x, y).$$

Now observe that $\mathcal{L}(\varphi_1) = L_2$, $\mathcal{L}(\varphi_2) = \mathcal{L}(\varphi_3) = \mathcal{L}(\varphi_4) = L_3$, and $\mathcal{L}(\varphi_5) = L_1$. Hence, if one of these relations is SpLog-selectable, the corresponding language is a SpLog-language, which contradicts our previous observations. ◀

# B Proofs from the Previous Paper

This section includes the proof of Theorem 27 in [14] (see the full version available at `http://ddfy.de/sci/spanners-conf.pdf`), which is used in the proof in Section A.8.2. In order to distinguish old material from new material, this section is separate from the "main appendix".

## B.1 Regex Formulas to EC$^{\mathsf{reg}}$ECreg

Before presenting the construction that is the main part of proof, we briefly consider a technical detail of functional regex formulas. On an intuitive level, functional regex formulas guarantee that in each parse tree, every variable is assigned exactly once (hence, $x\{\mathtt{a}\} \cdot x\{\mathtt{a}\}$ is not functional). Consequently, it seems reasonable to conjecture that, if a functional regex formula contains a subformula of the form $\alpha_1 \cdot \alpha_2$, $\mathsf{SVars}(\alpha_1) \cap \mathsf{SVars}(\alpha_2) = \emptyset$ must hold.

While this is true in general, $\emptyset$ can be used to construct regex formulas that disprove this conjecture, e. g. $x\{\mathtt{a}\}(x\{\emptyset\} \vee \mathtt{b})$. As $\emptyset$ is never part of a parse tree, this regex formula is functional.

In order to exclude these fringe cases and simplify the construction of EC$^{\mathsf{reg}}$-formulas, we introduce the following concept: A regex formula $\alpha$ is *$\emptyset$-reduced* if $\alpha = \emptyset$, or if $\alpha$ does not contain any occurrence of $\emptyset$. Using simple rewrite rules, we can observe the following.

▶ Claim 1. Given a regex formula $\alpha$, we can compute in polynomial time an $\emptyset$-reduced regex formula $\alpha_R$ with $[\![\alpha_R]\!] = [\![\alpha]\!]$.

Proof of Claim 1. In order to compute $\alpha_R$, it suffices to rewrite $\alpha$ according to the following rewrite rules:

1. $\emptyset^* \to \varepsilon$,
2. $(\hat{\alpha} \vee \emptyset) \to \hat{\alpha}$ and $(\emptyset \vee \hat{\alpha}) \to \hat{\alpha}$ for all regex formulas $\hat{\alpha}$,
3. $(\hat{\alpha} \cdot \emptyset) \to \emptyset$ and $(\emptyset \cdot \hat{\alpha}) \to \emptyset$ for all regex formulas $\hat{\alpha}$,
4. $x\{\emptyset\} \to \emptyset$ for all variables $x$.

As $\emptyset$ is never part of a parse tree, we can observe that for all regex formulas $\alpha$ and $\beta$, where $\beta$ is obtained by applying any number of these rewrite rules, $[\![\beta]\!] = [\![\alpha]\!]$ holds. Furthermore, one can use these rules to convert $\alpha$ into an equivalent and $\emptyset$-reduced $\alpha_R$ in polynomial time: If $\alpha$ is stored in a tree structure, it suffices to apply all applicable rules in bottom-up manner. ◀(for Claim 1)

This allows us to proceed to the main part of the proof. Recall that our goal is a procedure that, given a $\rho \in \mathsf{RGX}^{\mathsf{core}}$ with $\mathsf{SVars}(\rho) = \{x_1, \ldots, x_n\}$, constructs an EC$^{\mathsf{reg}}$-formula $\varphi_\rho(x_w, x_1^P, x_1^C, \ldots x_n^P, x_n^C)$ such that for all $w, w_1^P, w_1^C, \ldots, w_n^P, w_n^C \in \Sigma^*$,

$$(w, w_1^P, w_1^C, \ldots, w_n^P, w_n^C) \models \varphi_\rho$$

holds if and only if there is a $\mu \in P(w)$ with $w_k^P = w_{[1,i_k\rangle}$ and $w_k^C = w_{[i_k,j_k\rangle}$ for each $1 \leq k \leq n$, where $[i_k, j_k\rangle = \mu(x_k)$.

The most complicated part of this proof is the construction of EC$^{\mathsf{reg}}$-formulas from regex formulas.

▶ Claim 2. From every functional regex formula $\rho \in \mathsf{RGX}$ with $\mathsf{SVars}(\rho) = \{x_1, \ldots, x_n\}$, $n \geq 0$, we can construct in polynomial time an EC$^{\mathsf{reg}}$-formula $\varphi_\rho(x_w, x_1^P, x_1^C, \ldots, x_n^P, x_n^C)$ that realizes $[\![\rho]\!]$.

Proof of Claim 2. Due to Claim 1, we can assume without loss of generality that $\rho$ is $\emptyset$-reduced. We define $\varphi_\rho$ recursively as follows:

1. If $\rho$ does not contain any variables (i. e., $n = 0$), $\rho$ is a proper regular expression. Using canonical transformation techniques, we can construct in polynomial time a non-deterministic finite automaton $A$ with $\mathcal{L}(A) = \mathcal{L}(\rho)$, and we define
$$\varphi_\rho(x_w) := L_A(x_w).$$
Then $\varphi_\rho(w)$ is true if and only if $L_A(w)$ is true, which holds if and only if $w \in \mathcal{L}(A) = \mathcal{L}(\rho)$.
2. If $\rho$ contains variables, we assume that $\mathsf{SVars}(\rho) = \{x_1, \ldots, x_n\}$ with $n \geq 1$. By definition of regex formulas, no variable of $\rho$ may occur inside of a Kleene star. Hence, we can distinguish three cases:
   - **a.** $\rho = \rho_1 \vee \rho_2$, where $\rho_1, \rho_2$ are functional regex formulas with $\mathsf{SVars}(\rho_1) = \mathsf{SVars}(\rho_2) = \mathsf{SVars}(\rho)$. We define
$$\varphi_\rho(x_w, x_1^P, x_1^C, \ldots, x_n^P, x_n^C) := \big(\varphi_{\rho_1}(x_w, x_1^P, x_1^C, \ldots, x_n^P, x_n^C) \vee \varphi_{\rho_2}(x_w, x_1^P, x_1^C, \ldots, x_n^P, x_n^C)\big).$$
This formula is self explanatory: $\mu \in [\![\rho]\!](w)$ holds if and only if $\mu \in [\![\rho_1]\!](w)$ or $\mu \in [\![\rho_2]\!](w)$.
   - **b.** $\rho = \rho_1 \cdot \rho_2$, where $\rho_1, \rho_2$ are functional regex formulas with $\mathsf{SVars}(\rho_1) \cup \mathsf{SVars}(\rho_2) = \mathsf{SVars}(\rho)$ and $\mathsf{SVars}(\rho_1) \cap \mathsf{SVars}(\rho_2) = \emptyset$. Without loss of generality, we can assume $\mathsf{SVars}(\rho_1) = \{x_1, \ldots, x_m\}$ and $\mathsf{SVars}(\rho_2) = \{x_{m+1}, \ldots, x_n\}$ with $0 \leq m \leq n$. We define
$$\varphi_\rho(x_w, x_1^P, x_1^C, \ldots, x_n^P, x_n^C) := \exists y_1, y_2, z_{m+1}^P, \ldots, z_n^P \colon \Bigg( (x_w = y_1 \cdot y_2) \wedge \varphi_{\rho_1}(y_1, x_1^P, x_1^C, \ldots, x_m^P, x_m^C) \wedge \varphi_{\rho_2}(y_2, z_{m+1}^P, x_{m+1}^C, \ldots, z_n^P, x_n^C) \wedge \bigwedge_{m+1 \leq i \leq n} (x_i^P = y_1 \cdot z_i^P) \Bigg).$$
The idea behind this formula is as follows: As $\rho = \rho_1 \vee \rho_2$, whenever $[\![\rho]\!](w) \neq \emptyset$ holds, $w$ can be decomposed into $w = w_1 \cdot w_2$, where $w_1$ is parsed in $\rho_1$, and $w_2$ in $\rho_2$. We store these words in the variables $y_1$ and $y_2$, respectively. For all variables in $\mathsf{SVars}(\rho_1)$, the spans of the $\mu \in [\![\rho_1]\!](w_1)$ are also spans in $w$ (as $w_1$ is a prefix of $w$). Hence, we can use the results from $\rho_1$ unchanged. On the other hand, $[\![\rho_2]\!](w_2)$ determines spans in relation to $w_2$. Hence, each span $[i, j\rangle$ of $w_2$ corresponds to the span $[i + c, j + c\rangle$ of $w$, where $c := |w_1|$. The variables $z_i^P$ represent the start of the span with respect to $y_2$, and the conjunction of the equations $(x_i^P = y_1 \cdot z_i^P)$ converts these starts into spans with respect to $x_w$.
   - **c.** $\rho = x\{\hat{\rho}\}$ for some $x \in \{x_1, \ldots, x_n\}$, and $\hat{\rho}$ is a functional regex formula with $\mathsf{SVars}(\hat{\rho}) = \mathsf{SVars}(\rho) - \{x\}$. Without loss of generality, let $x = x_1$. We define
$$\varphi_\rho(x_w, x_1^P, x_1^C, \ldots, x_n^P, x_n^C) := \Big((x_1^P = \varepsilon) \wedge (x_1^C = x_w) \wedge \varphi_{\hat{\rho}}(x_w, x_2^P, x_2^C, \ldots, x_n^P, x_n^C)\Big).$$
This formula uses the fact that in this case, for each $\mu \in [\![\rho]\!](w)$, $\mu(x_1) = [1, |w| + 1\rangle$ must hold. This is encoded by $x_1^P = \varepsilon$ and $x_1^C = w$.

Now note that the construction of $\varphi_\rho$ follows the syntax of $\rho$ (no decisions beyond this are necessary), and the size of $\varphi_\rho$ is polynomial in the size of $\rho$. Hence, $\varphi_\rho$ can be computed in polynomial time. Finally, a straightforward induction on the structure of $\rho$ shows that $\varphi_\rho$ realizes $[\![\rho]\!]$. ◀(for Claim 2)

## B.2 Spanner Representations to EC$^{\mathsf{reg}}$

Now consider the case of an arbitrary core spanner representation $\rho \in \mathsf{RGX}^{\mathsf{core}}$ with $\mathsf{SVars}(\rho) = \{x_1, \ldots, x_n\}$, $n \geq 0$. We distinguish the following cases:

1. $\rho = \pi_Y \hat{\rho}$, with $Y = \mathsf{SVars}(\rho)$ and $\mathsf{SVars}(\hat{\rho}) \supseteq \mathsf{SVars}(\rho)$. Assume w.l.o.g. that $\mathsf{SVars}(\hat{\rho}) = \{x_1, \ldots, x_{n+m}\}$ with $m \geq 0$. We define

$$\varphi_\rho(x_w, x_1^P, x_1^C, \ldots, x_n^P, x_n^C) := \exists x_{n+1}^P, x_{n+1}^C, \ldots, x_{n+m}^P, x_{n+m}^C \colon \varphi_{\hat{\rho}}\left(x_w, x_1^P, x_1^C, \ldots, x_{n+m}^P, x_{n+m}^C\right)$$

2. $\rho = \zeta^=_{\vec{x}} \hat{\rho}$, with $\vec{x} \in (\mathsf{SVars}(\rho))^m$, $2 \leq m \leq n$, and $\mathsf{SVars}(\hat{\rho}) = \mathsf{SVars}(\rho)$. Assume w.l.o.g. that $\vec{x} = x_1, \ldots, x_m$. We define

$$\varphi_\rho(x_w, x_1^P, x_1^C, \ldots, x_n^P, x_n^C) := \left(\varphi_{\hat{\rho}}(x_w, x_1^P, x_1^C, \ldots, x_n^P, x_n^C) \wedge \bigwedge_{2 \leq i \leq m} (x_1^C = x_i^C)\right).$$

   Recall that $\zeta^=_{x_i, x_j}$ only checks whether $w_{\mu(x_i)} = w_{\mu(x_j)}$ holds, not whether $\mu(x_i) = \mu(x_j)$. This is equivalent to checking whether $x_i^C = x_j^C$ holds.

3. $\rho = (\rho_1 \cup \rho_2)$, with $\mathsf{SVars}(\rho_1) = \mathsf{SVars}(\rho_2) = \mathsf{SVars}(\rho)$. Let

$$\varphi_\rho(x_w, x_1^P, x_1^C, \ldots, x_n^P, x_n^C) := \left(\varphi_{\rho_1}\left(x_w, x_1^P, x_1^C, \ldots, x_n^P, x_n^C\right) \vee \varphi_{\rho_2}(x_w, x_1^P, x_1^C, \ldots, x_n^P, x_n^C)\right).$$

   In this case, we use $\mu \in [\![\rho]\!](w)$ if and only if $\mu \in [\![\rho_1]\!](w)$ or $\mu \in [\![\rho_2]\!](w)$.

4. $\rho = (\rho_1 \bowtie \rho_2)$ with $\mathsf{SVars}(\rho) = \mathsf{SVars}(\rho_1) \cup \mathsf{SVars}(\rho_2)$. We assume without loss of generality that $\mathsf{SVars}(\rho_1) = \{x_1, \ldots, x_l\}$ and $\mathsf{SVars}(\rho_2) = \{x_m, \ldots, x_n\}$ with $0 \leq l \leq n$ and $1 \leq m \leq n+1$. We define

$$\varphi_\rho(x_w, x_1^P, x_1^C, \ldots, x_n^P, x_n^C) := \left(\varphi_{\rho_1}(x_w, x_1^P, x_1^C, \ldots, x_l^P, x_l^C) \wedge \varphi_{\rho_2}(x_w, x_m^P, x_m^C, \ldots, x_n^P, x_n^C)\right).$$

   First, note that we have ensured that $\mathsf{SVars}(\rho_1) \cap \mathsf{SVars}(\rho_2) = \{x_l, \ldots x_m\}$. The definition of $\bowtie$ requires that $\mu \in [\![\rho]\!](w)$ holds if and only if there are $\mu_1 \in [\![\rho_1]\!](w)$ and $\mu_2 \in [\![\rho_2]\!](w)$ with $\mu_1(x_i) = \mu_2(x_i)$ for all $i \in \{l, \ldots, m\}$. For each of these variables $x_i$, $\varphi_{\rho_1}$ and $\varphi_{\rho_2}$ model the span with the same variables $x_i^P$ and $x_i^C$.

5. $\rho$ is a regex formula. This case is covered in Claim 2.

The formula $\varphi_\rho$ can be derived from $\rho$ without requiring further computation, and its size is polynomial in the size of $\rho$. Hence, $\varphi_\rho$ can be constructed in polynomial time. A straightforward induction on the structure of $\rho$ shows that $\varphi_\rho$ realizes $[\![\rho]\!]$.