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 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.ICDT.2017.3

1 Introduction

Fagin, Kimelfeld, Reiss, and Vansummeren [11] 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 α) 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, [11] introduced *regex formulas*, which are regular expressions with variables. Each variable x is connected to a subexpression α, and when α 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 [12] 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, [12] showed that every

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

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 [12] 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 [7, 6] 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 can be found only in the full version of the paper.

2 Preliminaries

Let Σ be a fixed finite alphabet of *(terminal) symbols*. Except when stated otherwise, we assume $|\Sigma| \geq 2$. Let Ξ be an infinite alphabet of *variables* that is disjoint from Σ. We use ε 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 Σ)* is a subset of $(\Sigma^*)^k$. Given a nondeterministic finite automaton (NFA) A (or a regular expression α), we use $\mathcal{L}(A)$ (or $\mathcal{L}(\alpha)$) to denote its language. In NFAs, we allow the use of ε-transitions (this model is also called ε-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 Σ, 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 μ 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 α^+ as shorthand for $\alpha \cdot \alpha^*$, and Σ 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 [12] 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 [23]. 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 α, 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 α. 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 α *functional* if $\mathsf{Ref}(\alpha) = \mathcal{R}(\alpha)$, and denote the set of all functional regex formulas by RGX.

In other words, $\mathcal{R}(\alpha)$ treats α 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 α is a functional, while β and γ are not (in fact, $\mathsf{Ref}(\alpha) = \mathsf{Ref}(\beta) = \emptyset$).

Like [11, 12], 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, clr projects ref-words to Σ). 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 α.

▶ **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 μ^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 [11]; 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 [11]. 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 Σ 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 δ 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 Σ 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 [11] (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 [11]).

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 $r_1 := \vdash_x \vdash_y \mathtt{a} \dashv_y \dashv_x$ and $r_2 := \vdash_y \vdash_x \mathtt{a} \dashv_x \dashv_y$. Both define the same $\mathtt{a}$-tuple μ (with $\mu(x) = \mu(y) = [1, 2\rangle$), although $r_1 \neq r_2$.

Fagin et al. [11] 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 Γ also includes $\dashv$, and extend 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 μ^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 VA as *v-automata*. An example for each type of v-automata can be found in Figure 1.

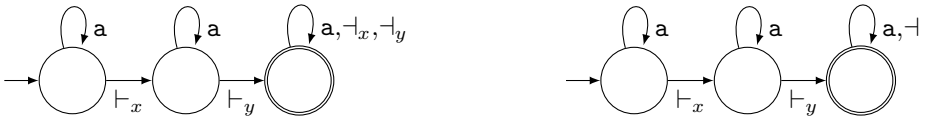

Figure 1 A vset-automaton A_{set} (left) and a vstk-automaton A_{stk} (right). Then $\mathsf{Ref}(A_{\mathsf{set}})$ consist of ref-words $r = \mathtt{a}^{i_1} \vdash_x \mathtt{a}^{i_2} \vdash_y \mathtt{a}^{i_3} \dashv_{z_1} \mathtt{a}^{i_4} \dashv_{z_2} \mathtt{a}^{i_5}$, with $i_1, \ldots, i_5 \geq 0$, $z_1, z_2 \in \{x, y\}$ and $z_1 \neq z_2$. Similarly, the ref-words from $\mathsf{Ref}(A_{\mathsf{stk}})$ are of the form $r = \mathtt{a}^{i_1} \vdash_x \mathtt{a}^{i_2} \vdash_y \mathtt{a}^{i_3} \dashv \mathtt{a}^{i_4} \dashv \mathtt{a}^{i_5}$, 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 μ 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 Σ^*. The *selection operator* ζ^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 [11] (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

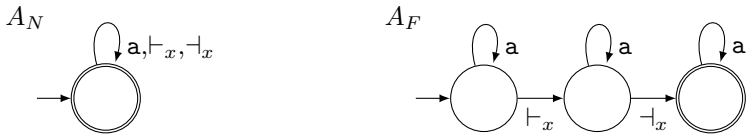

Figure 2 Two vset-automata A_N and A_F, which both define the universal spanner for the single variable x (cf. [11]) over the alphabet $\{\mathsf{a}\}$. As $\mathcal{R}(A_N)$ contains ref-words like $\mathsf{a}\dashv_x\mathsf{a}\vdash_x$ or $\mathsf{a}\vdash_x\mathsf{a}\vdash_x$, A_N is not functional. 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 O is a spanner algebra and C is a class of primitive spanner representations, then C^{O} denotes the set of all *spanner representations* that can be constructed by (repeated) combination of the symbols for the operators from 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 O.

Fagin et al. [11] 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 [11] defines 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 [11]: 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 μ, $\mu \in [\![A]\!](w)$ 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 NP-complete.*

The proof uses a basic reduction from the Hamiltonian path problem, which is NP-complete (cf. Garey and Johnson [13]). 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 [11], $[\![\mathsf{VA}_{\mathsf{stk}}]\!] \subset [\![\mathsf{VA}_{\mathsf{set}}]\!]$. 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 [11]). 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 (η_L, η_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 σ by defining $\sigma(x)$ for each $x \in \Xi$.

A substitution σ is a *solution* of a word equation (η_L, η_R) if $\sigma(\eta_L) = \sigma(\eta_R)$. The set of all variables in a pattern α 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* EC is obtained by combining word equations with $\wedge$, $\vee$, and existential quantification over variables. Formally, every word equation η is an EC-formula, and $\sigma \models \eta$ if σ is a solution of η. If φ_1 and φ_2 are 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 EC-formula φ and every $x \in \Xi$, $\psi := (\exists x\colon \varphi)$ is an 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 φ 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 $\varphi(x_1, \dots, x_k)$ to denote $\mathsf{free}(\varphi) = \{x_1, \dots, x_k\}$. Building on this, we also use $(w_1, \dots, w_k) \models \varphi(x_1, \dots, x_k)$ to denote $\sigma \models \varphi$ for the substitution σ that is defined by $\sigma(x_i) := w_i$, $1 \leq i \leq k$.

▶ **Example 14.** Consider the 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 EC-formula can be converted into a single word equation (cf. Karhumäki, Mignosi, and Plandowski [18]), and every $\mathsf{EC}^{\mathsf{reg}}$-formula into a single word equation with rational constraints (cf. Diekert [6]). 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 PSPACE-complete; but even for EC, showing the upper bound is by no means trivial (cf. [6, 8]). Note that negation is left out intentionally: Even the EC-fragment $\forall\exists^3$ (one universal over three existential variables) is undecidable (Durnev [9]).

3 SpLog: A Logic for Spanners

As shown by Freydenberger and Holldack [12], 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 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 φ, 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 SpLog-*formulas with main variable* W, $\mathsf{SpLog}(\mathsf{W})$, is the set of all safe $\varphi \in \mathsf{EC}^{\mathsf{reg}}$ such that
1. all word equations in φ are of the form $\mathsf{W} = \eta_R$, with $\eta_R \in ((\Xi - \{\mathsf{W}\}) \cup \Sigma)^*$,
2. for every subformula ψ of φ, $\mathsf{W} \in \mathsf{free}(\psi)$.

We also define the set of all 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 SpLog that exclusively defines constraints with regular expressions instead of NFAs.

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

▶ **Example 16.** Let $\varphi_1(\mathsf{W}; x) := \exists y, z_1, z_2 \colon (\mathsf{W} = yy \wedge \mathsf{W} = z_1 x z_2 \wedge \mathsf{C}_{\Sigma+}(x))$. Then φ_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, $\varphi_2(\mathsf{W}; x, y) := (\mathsf{W} = xx \vee \mathsf{W} = yyy)$ 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 ψ. We can compute in polynomial time a $\chi \in \mathsf{SpLog}(\mathsf{W})$ with $\chi \equiv (\varphi \wedge \psi)$.*

Proof. Let x_1, x_2 be new variables and define $\chi := \varphi \wedge \exists x_1, x_2 \colon \big((W = x_1 \cdot x \cdot x_2) \wedge \hat{\psi}\big)$, where $\hat{\psi}$ is obtained from ψ 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 ψ, $\chi \equiv (\varphi \wedge \psi)$ 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 [12]:

▶ **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 φ *realizes* P if, for all substitutions σ, $\sigma \models \varphi$ 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* φ if, for all substitutions σ, $\sigma \models \varphi$ 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* μ 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* μ. But such a spanner can directly be constructed from a spanner P that satisfies Definition 19, by joining P with a universal spanner (cf. [11]), 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 ρ_2 realizes ρ_1, and the size of ρ_2 is polynomial in the size of ρ_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 φ *realizes a spanner* P *modulo* ε if φ realizes a spanner $\hat{P}$ with $P(w) = \hat{P}(w)$ for all $w \in \Sigma^+$. In other words, φ realizes P on all inputs, except ε (where the behavior is undefined). Likewise, a *polynomial time conversion modulo* ε computes formulas that realize the spanners modulo ε. 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* ε *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 [12]. 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 ε or the first letter of W, which is the main reason that the construction only works modulo ε. 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 ε. But even if we insist on correctness on ε, 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* $\rho_f \in \mathsf{VA}^{\{\pi,\zeta^=,\cup,\times\}}_{\mathsf{set}}$ *or* $\rho_f \in \mathsf{VA}^{\{\pi,\zeta^=,\cup,\times\}}_{\mathsf{stk}}$*, where* ρ_f *is of polynomial size and every v-automaton in* ρ_f *is functional.*

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

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* $\varphi \in \mathsf{SpLog}$ *and a substitution* σ*, deciding* $\sigma \models \varphi$ *is NP-complete. For every fixed* $\varphi \in \mathsf{SpLog}$*, given a substitution* σ*, 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 [12] 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. [11] 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 φ_R such that $\vec{w} \models \varphi_R$ holds if and only if $\vec{w} \in R$, we know that we can use φ_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, [11] 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. [11], 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 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 φ. 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 $\mathsf{SpLog}(\mathsf{W})$-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 [11], 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 [18] 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 [21], 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:

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

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 α or β, 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 [12]; for a definition with ref-words, see Schmid [23] or the full version of this paper). For example, $x\{\Sigma^*\} \cdot \&x \cdot \&x$ generates the language of all www with $w \in \Sigma^*$.

As shown by Fagin et al. [11], core spanners cannot define all regex languages. But [12] 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 [12] can lead to an exponential blowup. The question whether a more efficient conversion is possible was left open in [12]. Using SpLog, we answer this positively:

▶ **Theorem 28.** *Given a vsf-regex α, 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 $\alpha_1, \dots, \alpha_n$, one can decide in PSPACE whether $\bigcap \mathcal{L}(\alpha_i) = \emptyset$ (by converting each α_i into a formula φ_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 $\varphi = \exists x_1, \dots, x_k \colon (\bigwedge_{i=1}^{m} \eta_i \wedge \bigwedge_{j=1}^{n} C_j)$, with $k, n \geq 0$, $m \geq 1$, where the η_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. [11] 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. [11]). 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 φ 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 $L/\mathrm{a} := \{w \mid wa \in L\}$. It is easily seen that the class of $\mathsf{EC}^{\mathsf{reg}}$-languages is closed under this operation, by using formulas like $\varphi_{/\mathrm{a}}(w) := \exists u \colon ((u = wa) \wedge \varphi(u))$. 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/a is a SpLog-language.*

This allows us to use Greibach's Theorem [15] to prove the following:

▶ **Proposition 32.** *The following conditions are equivalent:*
1. $\mathcal{L}(\mathsf{EC}^{\mathrm{reg}}) = \mathcal{L}(\mathsf{SpLog})$,
2. *Given* $\varphi \in \mathsf{EC}^{\mathrm{reg}}$, *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 $[\![\mathsf{EC}^{\mathrm{reg}}]\!] = [\![\mathsf{SpLog}]\!]$, an important question that is left open in the present paper (we define $[\![\mathcal{C}]\!] := \{[\![\varphi]\!] \mid \varphi \in \mathcal{C}\}$ for $\mathcal{C} \subseteq \mathsf{EC}^{\mathrm{reg}}$). 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 $\mathsf{EC}^{\mathrm{reg}}$, 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 [5], Karhumäki and Saarela [20]). 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 [5], Ilie and Plandowski [17]). 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 [12], 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 $\mathsf{EC}^{\mathrm{reg}}$, it is natural to ask whether this connection can be used to obtain inexpressibility results.

Karhumäki et al. [18] 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. [18]), but obviously SpLog-expressible. On the other hand, while $\mathsf{EC}^{\mathrm{reg}}$-inexpressibility results would be useful, to the author's knowledge, the only result in this direction is that every $\mathsf{EC}^{\mathrm{reg}}$-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 [10]), 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 [18] 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 [14]) with the observations on EC from [18] 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 [18] that we are going to use is based on the following definition by Karhumäki, Plandowski, and Rytter [19]:

▶ **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. [18, 19]). We use this definition in the following pumping result:

▶ **Theorem 36** (Karhumäki et al. [18])**.** *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. [11] (Theorem 4.21), $L_{\text{el}} := \{\mathtt{a}^i\mathtt{b}^i \mid i \geq 0\}$ is not expressible with core spanners (note that L_{el} is also used in [18] as an example application of Theorem 36). The length of this proof is roughly one page. Contrast this to the following: Assume that L_{el} is a SpLog-language. Then L_{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 $w = \varepsilon \cdot \mathtt{a}^{k+1} \cdot \mathtt{ab}^{k+2}$). Hence there exists a $u = \mathtt{a}^{k+2-j}\mathtt{b}^{k+2}$, $j > 0$, with $u \in L_{\text{el}}$. As $k + 2 - j < k + 2$, this is a contradiction.

From the inexpressibility of L_{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. [18] and Ilie [16] 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 Σ^*:*

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

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

Sadly, being limited to bounded languages also limits the applicability of this approach. For example, Ilie [16] 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. [11], 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 [22].

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