Noname manuscript No.
(will be inserted by the editor)

A Logic for Document Spanners

Dominik D. Freydenberger

Received: date / Accepted: date

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 formalize the query language AQL that is used in IBM's SystemT. As shown by Freydenberger and Holldack (ICDT 2016, ToCS 2018), 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 SpLog and core spanners. Consequences and applications include an alternative way of defining relations for spanners, a pumping lemma for core spanners, and insights into the relative succinctness of various classes of spanner representations and their connection to graph querying languages. We also briefly discuss the connection between SpLog with negation and core spanners with a difference operator.

Keywords Information extraction · document spanners · word equations · xregex · descriptional complexity · CRPQs with string equality

1 Introduction

Fagin, Kimelfeld, Reiss, and Vansummeren [13] introduced *document spanners* as a formal framework for information extraction in order to formalize the query language AQL that is used in SystemT, the information extraction engine of

Partly supported by Deutsche Forschungsgemeinschaft (DFG) under grant FR 3551/1-1. A preliminary version of this article appeared as [15].

D. D. Freydenberger
Loughborough University, Loughborough, United Kingdom, E-mail: ddfy@ddfy.de

IBM BigInsights [34]. On an intuitive level, document spanners can be viewed 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, [13] 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 π, union $\cup$, join $\bowtie$ (on spans), and string equality selection $\zeta^{=}$. Fagin et al. chose the term "core spanners" as these capture the core of the query language AQL, and thereby the core functionality of SystemT.

For example, assume the terminal alphabet Σ contains the usual ASCII symbols, Σ_{let} contains the lowercase letters `a` to `z`, and that we use ␣ to represent the space symbol. Now consider the following regex formula:

$$\alpha_{\mathsf{mail}}[x_{\mathsf{local}}, x_{\mathsf{domain}}] := \Sigma^{*} \text{␣} \, x_{\mathsf{local}}\{(\Sigma_{\mathsf{let}})^{+}\} \, \texttt{@} \, x_{\mathsf{domain}}\{(\Sigma_{\mathsf{let}})^{+}.(\Sigma_{\mathsf{let}})^{+}\} \text{␣} \, \Sigma^{*}$$

Then α_{mail} is a regex formula that matches (simplified) email addresses in the text. In every match, it stores the span of local part of the address (before the `@`) in the variable x_{local} and the span of the domain part (after the `@`) in the variable x_{domain}. Assume that the input word w contains each of the following two subwords exactly once:

$$u := \text{␣}\texttt{petra@example.com}\text{␣} \qquad v := \text{␣}\texttt{petra@example.edu}\text{␣}$$

Then the result of α_{mail} on w is a table that contains an entry that assigns the span of `petra` for the occurrence of u to x_{local} and the span of the corresponding `example.com` to x_{domain}. It also contains an element that assigns the spans of `petra` for the occurrence of v to x_{local} and the span for the corresponding `example.edu` to x_{domain}. Each additional occurrence of these words would produce another entry in the result table (and so would other parts of w that match). Using relational operators, core spanners can define more complicated queries, like the following:

$$\rho := \pi_{\emptyset}\zeta^{\neq}_{x_{\mathsf{domain}}, y_{\mathsf{domain}}}\zeta^{=}_{x_{\mathsf{local}}, y_{\mathsf{local}}}(\alpha_{\mathsf{mail}}[x_{\mathsf{local}}, x_{\mathsf{domain}}] \bowtie \alpha_{\mathsf{mail}}[y_{\mathsf{local}}, y_{\mathsf{domain}}])$$

Read from the inside out, ρ first builds two tables with spans for user and local parts of email addresses, as described above. These tables are then joined with $\bowtie$; and as the tables use different variables, this join acts like a cross product. After this, the string equality selection $\zeta^{=}_{x_{\mathsf{local}}, y_{\mathsf{local}}}$ ensures that in all remaining entries, the variables x_{local} and y_{local} describe the same word (but not necessarily at the same positions). Analogously, the string inequality selection ensures that the variables for the domain parts describe different words[1]. Finally, the projection turns ρ into a Boolean spanner (which returns only the empty tuple for "true", or the empty set for "false"). From our discussion, we conclude that ρ returns true if and only if the input text contains two email addresses that have the same local part, but different domains. So, if w contained the two example words u and v from above, ρ would return "true"; but if w consisted only of multiple occurrences of u, then ρ would return "false" (e. g., if $w = u^{99}$).

The main topic of this paper is a logic that captures core spanners. Freydenberger and Holldack [16] 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, [16] showed that every core spanner can be transformed 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 directly (as the encoding requires that the input word w of the spanner encodes the formula).

This paper further develops 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 prove 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, automata, or relational operators in the definition of core spanners. Moreover, this shows that the PSPACE upper bounds from [16] for deciding satisfiability and hierarchicality of regex formula based spanners apply to automata based spanners as well. We also 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.

In addition to these immediate uses and insights, the author also expects that SpLog will simplify future work on core spanners; in particular as the semantics of SpLog might be considered simpler than the semantics of core spanners and their variants. While the present paper mostly deals with core spanners (which use string equalities), we also introduce an alternative way of defining the semantics of the underlying regex formulas and v-automata using so-called ref-words. We shall see that this allows us to use various tools from automata theory with little or no extra effort.

[1] As we shall see in Section 5.1, string inequality selections can be used despite the fact that the definition of core spanners allows only equality selections.

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 [11,10] 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 reason to hope that SpLog can be applied to other models, like graph databases. In fact, we shall see that fragments of SpLog have natural counterparts in graph querying formalisms, if the latter are restricted to paths. As a related example of using $\mathsf{EC}^{\mathsf{reg}}$ for graph databases, Barceló and Muñoz [3] use a restricted class of $\mathsf{EC}^{\mathsf{reg}}$-formulas for which data complexity is also in NL.

The paper is structured as follows: Section 2 gives the definitions of $\mathsf{EC}^{\mathsf{reg}}$ and of spanners. Section 3 examines the notion of functional automata that provides additional context for the main result, as well as an efficient evaluation algorithm. Section 4 introduces SpLog (the main topic) and provides polynomial time transformations between SpLog-formulas and core spanners. We then examine properties of SpLog: Section 5 discusses how SpLog can be used to express relations and languages. In addition to offering an alternative way of defining relations for core spanners, this section also introduces and applies a normal form for SpLog, and gives an efficient conversion of a subclass of xregex (regular expressions with back-references) to SpLog. Section 6 examines what is not possible in SpLog: We use an EC-inexpressibility method to obtain the first general SpLog-inexpressibility method that does not rely on unary alphabets. We also briefly discuss separating SpLog from $\mathsf{EC}^{\mathsf{reg}}$. Section 7 explores connections between fragments of SpLog and graph querying languages, and uses this to obtain new restrictions on previous undecidability and descriptional complexity results for core spanners. Section 8 extends SpLog with negation, and connects the resulting logic $\mathsf{SpLog}^{\neg}$ to core spanners with difference. Section 9 concludes 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 models that this paper connects: word equations $\mathsf{EC}^{\mathsf{reg}}$ in Section 2.1, and document spanners in Section 2.2.

2.1 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 $x \in \Xi$, we have that $\psi := (\exists x\colon \varphi)$ is an EC-formula, and $\sigma \models \psi$ if there exists some $w \in \Sigma^*$ with $\sigma_{[x \to w]} \models \varphi$, where $\sigma_{[x \to w]}$ 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 $\mathsf{EC}^{\mathsf{reg}}$, the *existential theory of concatenation with regular constraints.* 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. Existential quantifiers may also range over multiple variables: In other words, 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. One could also argue in favor of $\mathsf{free}(C(x)) = \{x\}$; but for us, this question is moot, as our definitions in Section 4 will exclude this fringe case[2] the definitions in Section 4.

For all $\varphi \in \mathsf{EC}^{\mathsf{reg}}$, let $[\![\varphi]\!] := \{\sigma \mid \sigma \models \varphi\}$. For every $\mathcal{C} \subseteq \mathsf{EC}^{\mathsf{reg}}$, we define $[\![\mathcal{C}]\!] := \{[\![\varphi]\!] \mid \varphi \in \mathcal{C}\}$. 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

[2] More specifically, the distinction between these two definitions is only meaningful when dealing with constraints on variables that do not occur in word equations (like in formulas that consist only of constraint symbols). From an $\mathsf{EC}^{\mathsf{reg}}$ point of view, this are possible (although not of particular importance); but for spanners, these are not relevant.

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 $\sigma(x_i) := w_i$ for $1 \leq i \leq k$.

Example 2.1 Consider the EC-formula $\varphi_1(x, y, z) := \exists \hat{x}, \hat{y}\colon (x = z\hat{x} \land y = z\hat{y})$ and the $\mathsf{EC}^{\mathsf{reg}}$-formula $\varphi_2(x, y, z) := \exists \hat{x}, \hat{y}\colon (x = z\hat{x} \land y = z\hat{y} \land \mathsf{C}_{\Sigma^+}(z))$. 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 $\sigma \models \varphi_2$. $\diamond$

Word equations and EC have the same expressive power (cf. Choffrut and Karhumäki [6] or Karhumäki, Mignosi, and Plandowski [30]). More formally, for every EC-formula φ, one can construct a word equation η with $\mathsf{var}(\eta) \supseteq \mathsf{free}(\varphi)$, such that $\sigma \models \varphi$ if and only if there is a σ' with $\sigma' \models \eta$ and $\sigma'(x) = \sigma(x)$ for all $x \in \mathsf{free}(\varphi)$. This can directly be extended to convert any $\mathsf{EC}^{\mathsf{reg}}$-formula into a word equation with constraints (cf. Diekert [10]). For conjunctions, the construction is easily explained: Choose distinct $a, b \in \Sigma$. Hmelevskii's pattern pairing function is defined by $\langle \alpha, \beta \rangle := \alpha a \beta \alpha b \beta$. Then $(\alpha_L = \alpha_R) \land (\beta_L = \beta_R)$ holds if and only if $\langle \alpha_L, \beta_L \rangle = \langle \alpha_R, \beta_R \rangle$. This follows from a simple length argument, where the terminals a and b act as "barriers" that prevent unintended equalities (see Section 5.3 of [6] for details). The construction for disjunctions is similar, but it is also more involved and introduces new variables. Furthermore, converting alternating disjunctions and conjunctions may increase the size exponentially.

2.2 Document Spanners

2.2.1 Spanners and Primitive Spanner Representations

Let $w := a_1 a_2 \cdots a_n$ be a word over Σ, with $n \geq 0$ and $a_1, \dots, a_n \in \Sigma$. A *span of* w is an interval $[i, j\rangle$ with $1 \leq i \leq j \leq n + 1$. 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 2.2 Let $w := \mathsf{aabbcabaa}$. 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 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 $[1, 10\rangle$. $\diamond$

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 V is clear, 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 2.3 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 we use α^+ and Σ as shorthands for $\alpha \cdot \alpha^*$ and $\bigvee_{a \in \Sigma} a$, respectively.

Both syntax and semantics of regex formulas can be seen as special case of so-called *xregex*, a model that extends classical regular expressions with a repetition operator (see Section 5.3 for a brief and [16] for a more detailed discussion). In particular, both models define their syntax with parse trees, which is rather inconvenient for many of our proofs. Instead of using this definition, we present one that is based on the *ref-words* (short for *reference words*) of Schmid [41]. 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 2.4 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 $x \in \mathsf{SVars}(\alpha)$, we have $|r|_{\vdash_x} = 1$.

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, $\mathsf{Ref}(\alpha)$ consists of those words where each variable x is opened and closed exactly once.

Example 2.5 Define regex formulas $\alpha := (x\{\mathtt{a}\}y\{\mathtt{b}\}) \vee (y\{\mathtt{a}\}x\{\mathtt{b}\})$, $\beta_1 := x\{\mathtt{a}\} \vee y\{\mathtt{a}\}$, $\beta_2 := x\{\mathtt{a}\}x\{\mathtt{a}\}$, and $\beta_3 := (x\{\mathtt{a}\})^*$. Then α is a functional, while β_1 to β_3 are not. $\diamond$

Like [13,16], we adopt the convention that a regex formula is functional, unless we explicitly note otherwise[3]. Hence, without loss of generality, we assume that no variable binding $x\{\ \}$ occurs under a Kleene star $*$, and that no variable binding $x\{\}$ occurs inside a binding for the same variable.

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 end, 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 $g \in \Gamma$

[3] To be precise, the present paper and [16] follow the naming conventions of the conference version of [13]. In contrast to this, [13] uses the term "regex formula" exclusively for what we call "functional regex formula", and "variable regex" for what we call a "regex formula".

(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 each $r \in \mathsf{Ref}(\alpha, w)$ encodes a w-tuple μ^r that is consistent with α:

Definition 2.6 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 $\mu^r(x) := [i, j\rangle$, with $i := |\mathsf{clr}(r_1)| + 1$ and $j := |\mathsf{clr}(r_1 r_2)| + 1$. 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 2.7 Assume that $\mathtt{a}, \mathtt{b} \in \Sigma$. We define the functional regex formula

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

Let $w := \mathtt{baaba}$. Then $[\![\alpha]\!](w)$ consists of the tuples in the table to the left (we also picture w and its positions to the right):

$\mu(x)$	$\mu(y)$	$\mu(z)$
$[2,4\rangle$	$[3,3\rangle$	$[3,4\rangle$
$[2,5\rangle$	$[3,4\rangle$	$[4,5\rangle$
$[2,6\rangle$	$[3,5\rangle$	$[5,6\rangle$
$[3,5\rangle$	$[4,4\rangle$	$[4,5\rangle$
$[3,6\rangle$	$[4,5\rangle$	$[5,6\rangle$

b	a	a	b	a
1	2	3	4	5

As one example of an $r \in \mathsf{Ref}(\alpha, w)$, consider $r = \mathtt{b} \vdash_x \mathtt{a} \vdash_y \mathtt{a} \dashv_y \vdash_z \mathtt{b} \dashv_z \dashv_x \mathtt{a}$, which defines $\mu^r(x) = [2, 5\rangle$, $\mu^r(y) = [3, 4\rangle$, and $\mu^r(z) = [4, 5\rangle$, and corresponds to the following picture:

b	$\vdash_x$	a	$\vdash_y$	a	$\dashv_y \vdash_z$	b	$\dashv_z \dashv_x$	a
1		2		3		4		5

Although using ref-words is often convenient, it comes with a caveat. While $\mathsf{Ref}(\alpha_1) = \mathsf{Ref}(\alpha_2)$ implies $[\![\alpha_1]\!] = [\![\alpha_2]\!]$, the converse does not hold: For example, consider $\alpha_1 := x\{y\{\mathtt{a}\}\}$ and $\alpha_2 := y\{x\{\mathtt{a}\}\}$, and the ref-words $r_1 := \vdash_x \vdash_y \mathtt{a} \dashv_y \dashv_x$ and $r_2 := \vdash_y \vdash_x \mathtt{a} \dashv_x \dashv_y$ with $r_i \in \mathsf{Ref}(\alpha_i)$. Although $r_1 \neq r_2$, both define the same $\mathtt{a}$-tuple μ (with $\mu(x) = \mu(y) = [1, 2\rangle$). $\diamond$

It is easily seen that the definition of $[\![\alpha]\!]$ via ref-words is equivalent to the definition from [13]. Defining the semantics by 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 with little or no extra effort (see Section 3 in particular). Secondly, it generalizes naturally to vset- and vstk-automata, two models for defining spanners that we are going to discuss next. Both models were introduced in [13], using an equivalent definition of behavior that is based on runs. We begin with the first model.

Definition 2.8 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. Let $\mathsf{SVars}(A)$ denote the set of all $x \in V$ such that $\vdash_x$ or $\dashv_x$ occurs on a transition in δ.

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

An $r \in \mathcal{R}(A)$ is *valid* if, for every $x \in V$, $|r|_{\vdash_x} = |r|_{\dashv_x} = 1$, and $\vdash_x$ occurs to the left of $\dashv_x$. We define $\mathsf{Ref}(A)$, $\mathsf{Ref}(A, w)$, and $[\![A]\!]$ as for 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 [13] (there, opening and closing every variable exactly once is ensured 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 [13]).

Fagin et al. [13] also introduced another model, the *variable stack automaton (vstk-automaton)*. Its definition is almost identical to then vset-automaton; 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, we assume that Γ may include $\dashv$ instead of the symbols $\dashv_x$ (which type of closing symbol is used shall be clear from the context), and adapt clr by defining $\mathsf{clr}(\dashv) := \varepsilon$.

For every vstk-automaton A, we define $\mathcal{R}(A)$ and $\mathsf{SVars}(A)$ analogously to vset-automata. Accordingly, $\mathsf{Ref}(A)$ is the set of all valid $r \in \mathcal{R}(A)$, where r is valid if, for each $x \in V$, we have that $\vdash_x$ 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 $x \in V$, 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 $|r_2|_{\dashv} = \sum_{x \in V} |r_2|_{\vdash_x}$. This unique factorization allows us to obtain μ^r from $r \in \mathsf{Ref}(A)$ analogously to vset-automata.

We use *v-automaton* as general term that encompasses vset- and vstk-automata. Furthermore, we call a v-automaton *trim* if every state is reachable from its initial state, and the final state can be reached from every state. Each v-automaton can be turned straightforwardly into an equivalent trim v-automaton of the same type: Given some v-automaton A, let A_{trim} denote the automaton that is obtained from A by removing all states that are not reachable from the initial state, or from which the final state cannot be reached.

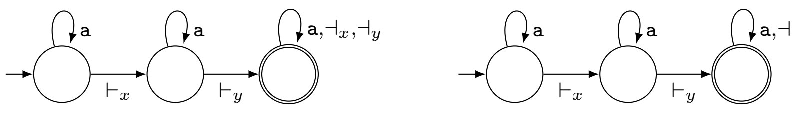


Fig. 1 A vset-automaton A_{set} (left) and a vstk-automaton A_{stk} (right). Then $\mathsf{Ref}(A_{\mathsf{set}})$ consist of ref-words $\mathtt{a}^{i_1}\vdash_x\mathtt{a}^{i_2}\vdash_y\mathtt{a}^{i_3}\dashv_{z_1}\mathtt{a}^{i_4}\dashv_{z_2}\mathtt{a}^{i_5}$, with $i_1,\ldots,i_5 \geq 0$, $z_1, z_2 \in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

Then $\mathcal{R}(A_{\mathsf{trim}}) = \mathcal{R}(A)$, which implies $[\![A]\!] = [\![A_{\mathsf{trim}}]\!]$. Thus, if A has n states and m transitions, then A_{trim} can be constructed in time $O(m+n)$ using a standard reachability analysis (e. g. by breadth-first search, see Cormen et al. [8]). For our purposes, this complexity is negligible; thus, we assume that every v-automaton is trim unless explicitly noted otherwise. We define the size of a v-automaton as the number transitions (for trim automata, the number of transitions dominates the number of states). Hence, assuming that Σ is fixed and keeping in mind that we consider trim automata by convention, the upper bound for the size of a v-automaton with n states and k variables is $O(kn^2)$.

Let $\mathsf{VA}_{\mathsf{set}}$ and $\mathsf{VA}_{\mathsf{stk}}$ be the classes of all trim vset-automata and all trim vstk-automata (respectively), and define $\mathsf{VA} := \mathsf{VA}_{\mathsf{set}} \cup \mathsf{VA}_{\mathsf{stk}}$. Examples for vset- and vstk-automata can be found in Figure 1.

Finally, observe that we can straightforwardly convert each regex formula α into a vset-automaton A with $\mathcal{R}(A) = \mathcal{R}(\alpha)$: First, we treat each $x\{\cdots\}$ as $\vdash_x \cdots \dashv_x$, thus interpreting α as regular expression for $\mathcal{R}(\alpha)$. Then, we transform this regular expression into a finite automaton. Finally, we ensure that the resulting automaton has exactly one final state (Definition 2.8 follows Fagin et al. [13] in requiring this). This allows us to use any algorithm that transforms a regular expression into an NFA, see Gruber and Holzer [26] for a survey that also considers complexity issues. An analogous observation can be made for the transformation to vstk-automata.

2.2.2 Spanner Algebras

In order to capture the expressive power of AQL, Fagin et al. [13] also defined the following spanner operators.

Definition 2.9 Let P, P_1, and P_2 be spanners. The algebraic operators *union*, *projection*, *natural join* and *selection* are defined as follows for all $w \in \Sigma^*$.

Union: If $\mathsf{SVars}(P_1) = \mathsf{SVars}(P_2)$, we define $(P_1 \cup P_2)$, the *union* of P_1 and P_2, by $\mathsf{SVars}(P_1 \cup P_2) := \mathsf{SVars}(P_1)$ and $(P_1 \cup P_2)(w) := P_1(w) \cup P_2(w)$.

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

Join: Let $V_i := \mathsf{SVars}(P_i)$ for $i \in \{1, 2\}$. Then $(P_1 \bowtie P_2)$, the *natural join* 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

$(P_1 \bowtie P_2)(w)$ 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: The *k-ary string equality selection operator* $\zeta^=$ is parameterized by k variables $x_1, \dots, x_k \in \mathsf{SVars}(P)$, written as $\zeta^=_{x_1,\dots,x_k}$. The *selection* $\zeta^=_{x_1,\dots,x_k} P$ is defined by $\mathsf{SVars}\left(\zeta^=_{x_1,\dots,x_k} P\right) := \mathsf{SVars}(P)$ and $\zeta^=_{x_1,\dots,x_k} P(w)$ is the set of all $\mu \in P(w)$ for which $w_{\mu(x_1)} = \cdots = w_{\mu(x_k)}$.

Take special note that join operates on spans, while selection compares the subwords of w that are described by the spans. Also observe that $P_1 \bowtie P_2$ 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 may write $\cap$ or $\times$ 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 primitive representations 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. [13] 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 $\mathsf{RGX}^{\mathsf{core}}$, $\mathsf{VA}^{\mathsf{core}}_{\mathsf{set}}$, $\mathsf{VA}^{\mathsf{core}}_{\mathsf{stk}}$, and $\mathsf{VA}^{\mathsf{core}}$.

3 On v-Automata

This section develops some basic insights on v-automata, which we use in Section 4 to provide further context for the main result: Section 3.1 introduces and examines functional v-automata, while Section 3.2 examines the relative succinctness of different classes of v-automata.

3.1 Functionality and Evaluation of v-Automata

We begin with a short observation on the complexity of the evaluation of v-automata, namely that even on the empty word, evaluation is hard.

Lemma 3.1 *Given $A \in \mathsf{VA}$, deciding whether $[\![A]\!](\varepsilon) \neq \emptyset$ is NP-hard.*

Proof. We show NP-hardness by reduction from the directed Hamiltonian path problem (see e. g. Garey and Johnson [21]), which is defined as follows: Given a directed graph $G = (V, E)$, does G contain a Hamiltonian path? A Hamiltonian path is a sequence $(i_1, \dots, i_n)$ with $n = |V|$, $i_1, \dots, i_n \in V$, and $(i_j, i_{j+1}) \in E$ for all $1 \leq j < n$, 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 $A \in \mathsf{VA}_{\mathsf{set}}$ such that $[\![A]\!](\varepsilon) \neq \emptyset$ if and only if G

contains a Hamiltonian path. Assume that $V = \{1, \ldots, n\}$ for some $n \geq 1$. We shall define A with $\mathsf{SVars}(A) = \{x_1, \ldots, x_n\}$. Let $A := (Q, q_0, q_f, \delta)$, where $Q := \{q_0, q_f\} \cup \{q_i \mid 1 \leq i \leq n\}$, and δ is defined as follows:

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

The intuition behind the automaton 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, the reduction represents each edge $(i,j) \in E$ as a transition from q_i to q_j that is labeled with $\vdash_{x_j}$. Finally, at any point, A can change to the final state by reading any $\dashv_{x_j}$. It then finishes by closing all remaining variables.

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

Moreover, if $r \in \mathsf{Ref}(A)$, then each $\vdash_{x_i}$ has to occur exactly once in r, which means that the path $(i_1, \ldots, i_k)$ 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 Σ, $\mathsf{Ref}(A) = \mathsf{Ref}(A, \varepsilon)$. Hence, $\mathsf{Ref}(A, \varepsilon) \neq \emptyset$ if and only if G contains a Hamiltonian path. As the Hamiltonian path problem is NP-complete, 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 $\dashv$. □

Furthermore, note that for every set of variables V, there exists only one possible (V, ε)-tuple μ (namely $\mu(x) = [1, 1\rangle$ for all $x \in V$). Hence, Lemma 3.1 also establishes the following.

Corollary 3.2 *Given $A \in \mathsf{VA}$, $w \in \Sigma^*$, and a (V, w)-tuple μ, deciding whether $\mu \in [\![A]\!](w)$ is NP-hard.*

The proof of Lemma 3.1 uses that the semantics of v-automata ensure that every variable is opened and closed exactly once (or, in ref-word terminology, it uses that the semantics are defined only by valid ref-words, instead of the full ref-language). This raises the question whether these problems become tractable if we restrict the automata analogously.

Although [13] defines RGX as the set of functional regex formulas, no such notion is introduced for v-automata. But there is a natural way of defining this: First, consider that every match of a functional regex formula guarantees that every variable is assigned exactly once (in contrast to non-functional regex formulas like $x\{\mathsf{a}\}x\{\mathsf{a}\}$ and $x\{\mathsf{a}\} \vee y\{\mathsf{a}\}$, which assign variables twice or not at all). Using ref-word terminology, this means that $\mathsf{Ref}(\alpha, w)$ can be derived directly from $\mathcal{R}(\alpha)$, as this language contains only valid ref-words.

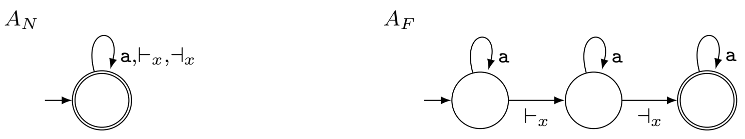


Fig. 2 Two vset-automata A_N and A_F, which both define the universal spanner for the single variable x (cf. [13]) over the alphabet $\{\mathsf{a}\}$. As $\mathcal{R}(A_N)$ contains ref-words like $\mathsf{a}\dashv_x\mathsf{a}\vdash_x$ or $\mathsf{a}\vdash_x\mathsf{a}\vdash_x$, A_N is not functional. 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.

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 [13]: Translated to these semantics, a v-automaton A is functional if every path from q_0 to q_f describes an accepting run. At the end of Section 2.2.1, we discussed that transformations of regular expressions into finite automata can be used to transform a regex formula α into a vset-automaton A with $\mathcal{R}(A) = \mathcal{R}(\alpha)$. Hence, every functional regex formula can be transformed into an $\mathcal{R}$-equivalent functional vset-automaton. Again, analogous observations can be made for vstk-automata.

While v-automata in general have to keep track of the used variables, functional v-automata store this information implicitly in their states. We formalize this in the following definition.

Definition 3.3 Let $A \in \mathsf{VA}$ be functional with $A = (Q, q_0, q_f, \delta)$. For every $q \in Q$, we define

- a set O_q that contains the variables that have been opened when A is in state q, and
- if A is a vset-automaton, a set C_q that contains the variables that have been closed when A is in state q; or,
- if A is a vstk-automaton, a number N_q that is the number of variables that have been closed when A is in state q.

More formally and using ref-words, we can define these as follows.

$$\begin{aligned}
O_q &:= \{x \in \mathsf{SVars}(A) \mid q \in \delta^*(q_0, r) \text{ for some } r \in (\Sigma \cup \Gamma)^* \text{ with } |r|_{\vdash_x} = 1\},\\
C_q &:= \{x \in \mathsf{SVars}(A) \mid q \in \delta^*(q_0, r) \text{ for some } r \in (\Sigma \cup \Gamma)^* \text{ with } |r|_{\dashv_x} = 1\},\\
N_q &:= |r|_{\dashv} \text{ for some } r \in (\Sigma \cup \Gamma)^* \text{ with } q \in \delta^*(q_0, r).
\end{aligned}$$

It is an important feature of functional v-automata that any ref-word that leads from q_0 to q can be used to define O_q and C_q (or O_q and N_q).

Lemma 3.4 *Let $A \in \mathsf{VA}$ be functional with $A = (Q, q_0, q_f, \delta)$ and let $q \in Q$. For all ref-words $r_1, r_2 \in (\Sigma \cup \Gamma)^*$ with $q \in \delta^*(q_0, r_1) \cap \delta^*(q_0, r_2)$, we have:*

1. *$|r_1|_{\vdash_x} = |r_2|_{\vdash_x}$ for all $x \in \mathsf{SVars}(A)$, and,*
2. *if A is a vset-automaton, $|r_1|_{\dashv_x} = |r_2|_{\dashv_x}$ for all $x \in \mathsf{SVars}(A)$, or*

3. if A is a vstk-automaton, $|r_1|_{\dashv} = |r_2|_{\dashv}$.

Proof. We only prove the first claim, the others follow analogously. Assume there exist ref-words $r_1, r_2 \in (\Sigma \cup \Gamma)^*$ such that $|r_1|_{\vdash_x} \neq |r_2|_{\vdash_x}$ for some $x \in \mathsf{SVars}(A)$, and there is a state $q \in \delta^*(q_0, r_1) \cap \delta^*(q_0, r_2)$.

Recall that A is trim by definition of VA. Hence, there exist $s_1, s_2 \in (\Sigma \cup \Gamma)^*$ with $q_f \in \delta^*(q, s_i)$. Thus, for all $i, j \in \{1, 2\}$, we have that $(r_i \cdot s_j) \in \mathcal{R}(A)$, which leads to $(r_i \cdot s_j) \in \mathsf{Ref}(A)$, as A is functional.

Therefore, every $r_i \cdot s_j$ must be valid, which implies $|r_i \cdot s_i|_{\vdash_x} = 1$. As a consequence, $|r_i|_{\vdash_x} \in \{0, 1\}$. Combining this with our initial assumption of $|r_1|_{\vdash_x} \neq |r_2|_{\vdash_x}$, we conclude that one of the ref-words r_1 and r_2 contains exactly one occurrence of $\vdash_x$, while the other ref-word contains no occurrence of $\vdash_x$. Assume without loss of generality that $|r_1|_{\vdash_x} = 1$ and $|r_2|_{\vdash_x} = 0$. As $r_2 \cdot s_2$ is valid, the latter implies $|s_2|_{\vdash_x} = 1$. Hence, $|r_1 \cdot s_2|_{\vdash_x} = 2$, which means that the ref-word $r_1 \cdot s_2$ is invalid. Contradiction. □

Hence, Lemma 3.4 allows us to compute all O_q and all C_q (or O_q and N_q) by choosing any ref-word that takes A from q_0 to q. This provides us with the following functionality test that we shall also use as part of an evaluation algorithm for functional v-automata.

Lemma 3.5 *There is an algorithm that, given $A \in \mathsf{VA}$ with m transitions, and k variables, decides whether A is functional in time $O(km)$.*

If A is functional, the algorithm also computes all O_q and all C_q (if $A \in \mathsf{VA}_{\mathsf{set}}$) or all O_q and all N_q (if $A \in \mathsf{VA}_{\mathsf{stk}}$) as defined in Definition 3.3.

Proof. Let $A = (Q, q_0, q_f, \delta)$ be a v-automaton. We first discuss the algorithm for vset-automata, and then how it can be adapted to vstk-automata.

Algorithm for vset-automata: A pseudo-code representation of this algorithm can be found in Algorithm 1. We know A is trim by definition of $\mathsf{VA}_{\mathsf{set}}$. Hence, every state can be reached from q_0, and q_f can be reached from every state.

The algorithm tries to find a state q that violates Lemma 3.4. To do so, it inductively constructs all O_q and all C_q, while looking for a transition that causes these sets to be inconsistent.

We start by defining $O_{q_0} := C_{q_0} := \emptyset$, and declaring all sets O_q and C_q with $q \neq q_0$ as undefined. In the main loop, the algorithm picks a state $p \in Q$ that has not been picked before and for which O_p and C_p are defined. It then iterates over all transitions from p. For each such transition from p to some state $q \in Q$ with some label $\lambda \in (\Sigma \cup \Gamma \cup \{\varepsilon\})$, we know that a functional automaton must satisfy the following conditions that depend on λ:

- if $\lambda \in (\Sigma \cup \{\varepsilon\})$, then $O_q = O_p$ and $C_q = C_p$ must hold,
- if $\lambda = \vdash_x$, then $x \notin O_p$, $O_q = O_p \cup \{x\}$, and $C_q = C_p$ must hold,
- if $\lambda = \dashv_x$, then $x \in O_p$, $x \notin C_p$, $O_q = O_p$, and $C_q = C_p \cup \{x\}$ must hold.

In each case, the conditions describe that the sets for q are correct successors to the sets for p after using this transition. For the variable transitions, the conditions also ensure that each variable is opened or closed only once, and that a variable can only be closed if it has been opened.

If the current transition is a variable transition (i. e., $\lambda \in \{\vdash_x, \dashv_x\}$ for some $x \in \mathsf{SVars}(A)$), the algorithm first checks either whether $x \notin O_p$ (if $\lambda = \vdash_x$), or whether $x \in O_p$ and $x \notin C_p$ (if $\lambda = \dashv_x$). If this check fails, the algorithm terminates and declares that A is not functional (as q contradicts Lemma 3.4).

If this check succeeds, or if the transition is not a variable transition, the algorithm distinguishes two cases:

- If O_q and C_q are undefined, it defines them according to the respective condition and continues.
- If O_q and C_q are defined, the algorithm checks whether the sets satisfy the respective condition. If this check fails, the algorithm terminates and declares that A is not functional (like above, we know that q contradicts Lemma 3.4). Otherwise, it continues.

If A has not been declared as not functional, the algorithm then proceeds to the next transition for p (or the next iteration of the main loop).

After the main loop has finished without declaring A as not functional, we know that all transitions of A result in consistent sets O_q and C_q. Finally, the algorithm declares A to be functional if and only if $C_{q_f} = \mathsf{SVars}(A)$. This is correct for the following reason: If there is an $x \in (\mathsf{SVars}(A) - C_{q_f})$, we know that $|r|_{\dashv_x} = 0$ for all $r \in \mathcal{R}(A)$. Hence, $\mathcal{R}$ contains invalid words, which means that A is not functional.

On the other hand, $C_{q_f} = \mathsf{SVars}(A)$ implies $O_{q_f} = \mathsf{SVars}(A)$, as the conditions above ensure that $C_q \subseteq O_q$ for all $q \in Q$. Furthermore, the conditions also ensure that each variable is opened and closed exactly once. This allows us to conclude that for all $x \in \mathsf{SVars}(A)$, every $r \in \mathcal{R}(A)$ contains each of $\vdash_x$ and $\dashv_x$ exactly once, and in the right order. Hence, $\mathcal{R}(A) = \mathsf{Ref}(A)$, which means that A is functional, and we can output the sets O_q and C_q for all $q \in Q$.

All that remains is to verify the upper bound on the running time: The main loop and the included iterations over the transitions touch each of the m transitions exactly once. For each transition, we can perform the checks on the sets in time $O(k)$. This yields a total time of $O(km)$.

Algorithm for vstk-automata: This requires only minor modifications: We define $N_{q_0} := 0$, and N_q defaults to undefined for each $q \neq q_0$. The conditions for transitions from p to q with label λ are as follows:

- if $\lambda \in (\Sigma \cup \{\varepsilon\})$, then $O_q = O_p$ and $N_q = N_p$ must hold,
- if $\lambda = \vdash_x$, then $x \notin O_p$, $O_q = O_p \cup \{x\}$, and $N_q = N_p$ must hold,
- if $\lambda = \dashv$, then $|N_p| < |O_p|$, $O_q = O_p$, and $N_q = N_p + 1$ must hold.

The only noteworthy change here is in the last condition: There, we can only process $\dashv$ if the number of variables that has already been closed is smaller than the number of variables that has been opened. Apart from that, the algorithm

Algorithm 1: Functionality test for vset-automata

```
Input: trim vset-automaton $A = (Q, \delta, q_0, q_f)$ with variables $V$
Output: False if $A$ is not functional, True and all sets $O_q$ and $C_q$ if $A$ is functional.
$O_{q_0} := \emptyset$;
$C_{q_0} := \emptyset$;
forall $q \in Q - \{q_0\}$ do
    $O_q :=$ undefined;
    $C_q :=$ undefined;
Seen $:= \{q_0\}$;
ToDo $:= \{q_0\}$;
while ToDo $\neq \emptyset$ do
    choose and remove any $q$ from ToDo;
    foreach $p$ such that $p \in \delta(q, a)$ for some $a \in \Sigma$ do
        if $p \in$ Seen then
            if $O_p \neq O_q$ or $C_p \neq C_q$ then return False;
        else
            add $p$ to Seen and to ToDo;
            $O_p := O_q$;
            $C_p := C_q$;
    foreach $p$ such that $p \in \delta(q, \vdash_x)$ for some $x \in V$ do
        if $x \in O_q$ then return False;
        if $p \in$ Seen then
            if $O_p \neq (O_q \cup \{x\})$ or $C_p \neq C_q$ then return False;
        else
            add $p$ to Seen and to ToDo;
            $O_p := (O_q \cup \{x\})$;
            $C_p := C_q$;
    foreach $p$ such that $p \in \delta(q, \dashv_x)$ for some $x \in V$ do
        if $x \notin O_q$ or $x \in C_q$ then return False;
        if $p \in$ Seen then
            if $O_p \neq O_q$ or $C_p \neq (C_q \cup \{x\})$ then return False;
        else
            add $p$ to Seen and to ToDo;
            $O_p := O_q$;
            $C_p := (C_q \cup \{x\})$;
if $C_{q_f} \neq V$ then
    return False
else
    return True and all sets $O_q$ and $C_q$
```

proceeds as for vset-automata, with the final check whether $|N_{q_f}| = |\mathsf{SVars}(A)|$. Analogously to the vset-case, this holds only if $O_{q_f} = \mathsf{SVars}(A)$. □

Recall that we showed in Lemma 3.1 and Corollary 3.2 suggest that evaluation of v-automata *in general* is NP-hard. But for *functional* v-automata, we can use the information that is encoded in the O_q and C_q (or O_q and N_q) for an efficient evaluation algorithm. In other words, non-functionality is the only source of intractability for v-automata evaluation.

Lemma 3.6 *Given $w \in \Sigma^*$, a functional $A \in \mathsf{VA}$, and a $(\mathsf{SVars}(A), w)$-tuple μ, we can decide in polynomial time whether $\mu \in [\![A]\!](w)$.*

Proof. We first show the vset-automata case; the construction for vstk-automata only requires some minor modifications and is given at the end of the proof. Let $A = (Q, q_0, q_f, \delta)$ be a functional vset-automaton. Now, we need to keep in mind that, for every $w \in \Sigma^*$, multiple ref-words r can define the same $(\mathsf{SVars}(A), w)$-tuple μ^r. For example, if $\mu(x) = \mu(y)$, the corresponding ref-word can contain e. g. $\vdash_x\vdash_y\dashv_x\dashv_y$ or $\vdash_y\dashv_y\vdash_x\dashv_x$ (or any other arrangement that opens and closes each variable in the right order). To deal with this partial commutativity, we represent μ as a sequences of words $w_0, \ldots, w_n \in \Sigma^*$ and sequence of sets $M_1, \ldots, M_n \subseteq \Gamma$ 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 $0 < i < n$,
3. the sets $M_1, \ldots, M_n$ are non-empty and pairwise disjoint,
4. $\bigcup_{i=1}^{n} M_i = \{\vdash_x, \dashv_x \mid x \in \mathsf{SVars}(A)\}$,
5. $\mu(x) = [o, c\rangle$ if and only if there exist $1 \leq i \leq j \leq n$ with $\vdash_x \in M_i$, $\dashv_x \in M_j$, $o = |w_0 \cdots w_{i-1}| + 1$, and $c = |w_0 \cdots w_{j-1}| + 1$.

Intuitively, the combined sequence $w_0, M_1, w_1, \ldots, M_n, w_n$ describes how A has to match μ to w, where successive variable transitions are considered commutative. The words w_i describe the how A consumes the input, and the sets M_i describe how A acts on variables. Hence, the sequence captures how A alternates between both types of behavior.

As a consequence, 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 $v_i \in \Gamma^+$ such that $r = w_0(v_1 w_1) \cdots (v_n w_n)$. As we require $w_i \neq \varepsilon$ and $S_i \neq \emptyset$, every pair μ and w defines a unique pair of sequences $w_0, \ldots, w_n$ and $M_1, \ldots, M_\ell$.

We now simulate all possible r with $\mu^r = \mu$ using a generalization of the on-the-fly computation of the powerset construction (for the simulation of NFAs with DFAs). More specifically, the algorithm shall construct a sequence of sets $S_0, T_1, S_1, \ldots, T_n, S_n \subseteq Q$, where each S_i describes the states that A can have after processing $w_0, (M_1, w_1), \ldots, (M_i, w_i)$, while T_i describes the states that can be reached by processing $(w_0, M_1), \ldots, (w_{i-1}, M_i)$.

In order to ensure that the T_i are computed correctly, we also define

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

for all $1 \leq i \leq \ell$, 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 $w_0(M_1 w_1) \cdots (M_i w_i)$. This necessarily results in $O_n = C_n = \mathsf{SVars}(A)$.

We now define $S_0 := \delta^*(q_0, w_0)$. The algorithm then iterates the following loop for i from 1 to n:

1. Let T_i be the set of all states $q \in Q$ such that
   (a) $O_q = O_i$ and $C_q = C_i$, and

(b) there exists a state $p \in S_{i-1}$ such that q can be reached from p using only ε-transitions and variable-transitions.
2. Let $S_i := \bigcup_{p \in T_i} \delta^*(p, w_i)$.

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 $\mu \in [\![A]\!](w)$; and this is clearly possible in polynomial time (recall that due to Lemma 3.5, we can precompute the sets O_q and C_q for all $q \in Q$ in polynomial time).

vstk-automata: For vstk-automata, instead of storing different $\dashv_x$ in the sets M_i, and using these to compute C_i for every step i, we compute sets N_i that determine how many variables have been closed.

We also have to refine the computation of the T_i to account for a special case of opening variables: Due to the stack behavior, we can encounter cases where two variables are opened in the same M_i, but closed at different times. Those variables are not commutative within M_i. Hence, we define the partial order $\prec_\mu$ on $\mathsf{SVars}(A)$ such that $x \prec_\mu y$ if $\mu(x) = [k, m\rangle$ and $\mu(y) = [k, n\rangle$ with $m < n$. In addition to the criteria that hold for vset-automata, the reachability analysis that computes T_i now may only use transitions from some state p' to some state q' with label $\vdash_y$ if $x \in O_{p'}$ for all $x \prec_\mu y$.

Apart from that, we 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 O_i and N_i determine which states are viable destinations. Clearly, $\prec_\mu$ can be computed in polynomial time from μ. □

This approach was used by Freydenberger, Kimelfeld, and Peterfreund [17] to develop a polynomial delay algorithm for regular spanners.

3.2 Relative Succinctness of v-Automata

Our next goal is to compare the succinctness of functional and general v-automata, as well as that of vstk- and vset-automata. To this end, we introduce a lemma that allows us to treat certain v-automata as NFAs that accept ref-words. Note that the result applies regardless of whether the ref-words close variables by name with $\dashv_x$ or by stack with $\dashv$. But as a convention, we shall only apply the following lemma to two ref-words if either both of them close variables by name or both of them close variables by stack.

Lemma 3.7 *For a finite $V \subset \Xi$, consider any valid $r \in (\Sigma \cup \Gamma_V)^*$ that contains no subword from Γ_V^2. Then for every valid $\hat{r} \in (\Sigma \cup \Gamma_V)^*$ with $\mathsf{clr}(\hat{r}) = \mathsf{clr}(r)$ that closes variables in the same way as r, we have that $\mu^{\hat{r}} = \mu^r$ implies $\hat{r} = r$.*

Proof. Every valid $r \in (\Sigma \cup \Gamma_V)^*$ that contains no subword from Γ_V^2 has a unique factorization $r = w_0(v_1 w_1) \cdots (v_{2k} w_{2k})$ with $v_i \in \Gamma_V$, $w_0, w_{2k} \in \Sigma^*$, and $w_1, \ldots, w_{2k-1} \in \Sigma^+$. Hence, for all $x \in V$ and $[i_x, j_x\rangle := \mu^r(x)$, we have $i_x \neq j_x$; and for all $y \in (V - \{x\})$ and $[i_y, j_y\rangle := \mu^r(y)$, we know i_x, j_x, i_y, j_y are pairwise distinct.

Now assume that there is a valid $\hat{r} \in (\Sigma \cup \Gamma_V)^*$ with $\mathsf{clr}(\hat{r}) = \mathsf{clr}(r)$ and $\mu^{\hat{r}} = \mu^r$. We first observe that $\hat{r}$ contains no factor from Γ_V^2. Otherwise, we would have $\mu^{\hat{r}} \neq \mu^r$, as there would be some $x \in V$ where $i_{\hat{x}} = j_{\hat{x}}$ for $[i_{\hat{x}}, j_{\hat{x}}\rangle := \mu^{\hat{r}}(x)$, or there is some $y \in (V - \{x\})$ such that $i_{\hat{x}}, j_{\hat{x}}, i_{\hat{y}}, j_{\hat{y}}$ are not pairwise distinct for $[i_{\hat{y}}, j_{\hat{y}}\rangle := \mu^{\hat{r}}(y)$.

Thus, $\hat{r}$ can be factorized into $\hat{r} = \hat{w}_0(\hat{v}_1\hat{w}_1)\cdots(\hat{v}_{2k}\hat{w}_{2k})$, analogously to r. By comparing the factorizations of r and $\hat{r}$ from left to right, we observe that $\hat{w}_i = w_i$ and $\hat{v}_j = v_j$ has to hold for all i and j. Otherwise, we would obtain a contradiction to $\mu^{\hat{r}} = \mu^r$ or $\mathsf{clr}(\hat{r}) = \mathsf{clr}(r)$. We conclude $\hat{r} = r$. □

Lemma 3.7 provides us with a sufficient criterion for ref-words r that uniquely define μ^r. This allows us to identify v-automata that can be treated as NFAs. In particular, we shall use the following result by Birget [4] (although the proof in [4] refers only to NFAs without ε-transitions, it directly generalizes to those with ε-transitions).

Lemma 3.8 (Birget [4]) *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.

Now we are ready to compare functional and general v-automaton. The author considers it no surprise that standard automata techniques allow us to transform every vset- or vstk-automaton into an equivalent functional v-automaton of the same type; but this may result in an exponential number of states. While combining Lemma 3.1 with Lemma 3.6 already suggests that this conversion is not possible in polynomial *time* (unless the number of variables is bounded, or $\mathsf{P} = \mathsf{NP}$), we also show matching exponential *size* bounds.

Proposition 3.9 *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 $A_{\mathsf{fun}} \in \mathsf{VA}_s$ 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 $A_{\mathsf{fun}} \in \mathsf{VA}_s$ has at least $f_s(k)$ states.*

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 $A = (Q, q_0, q_f, \delta)$ with $k \geq 1$ variables. Our goal is to construct a functional vset-automaton A_{fun} with $3^k|Q|$ states and $[\![A_{\mathsf{fun}}]\!] = [\![A]\!]$. The main idea is to intersect A with a functional vset-automaton that keeps track of the sets O_q and C_q for all $q \in Q$ (Definition 3.3). Formally, we associate each state of A_{fun} 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 $\vdash_x$ has not been read,

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

Let S be the set of all such functions. Observe that $|S| = 3^k$. We now define $A_{\mathsf{fun}} := (Q_{\mathsf{fun}}, q_0^{\mathsf{fun}}, q_f^{\mathsf{fun}}, \delta_{\mathsf{fun}})$ in the following way:

– $Q_{\mathsf{fun}} := Q \times S$,
– $q_0^{\mathsf{fun}} := (q_0, s_0)$, where s_0 is defined by $s_0(x) = \mathtt{w}$ for all $x \in \mathsf{SVars}(A)$,
– $q_f^{\mathsf{fun}} := (q_f, s_f)$, where s_f is defined by $s_f(x) = \mathtt{c}$ for all $x \in \mathsf{SVars}(A)$,
– $\delta_{\mathsf{fun}}((p, s), a) := \{(q, s) \mid q \in \delta(p, a)\}$ for $a \in (\Sigma \cup \{\varepsilon\})$ and $(p, s) \in Q_{\mathsf{fun}}$,
– for all $(p, s) \in Q_{\mathsf{fun}}$ and all $x \in \mathsf{SVars}(A)$, let

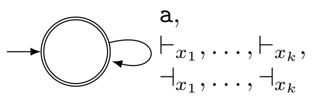
$$\delta_{\mathsf{fun}}((p,s),\vdash_x) := \begin{cases} \emptyset & \text{if } s(x) \neq \mathtt{w}, \\ \{(q,t_o) \mid q \in \delta(p,\vdash_x)\} & \text{if } s(x) = \mathtt{w}, \end{cases}$$

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

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_{fun} is correct and functional, note that it simulates A, while the definition of δ_{fun} ensures that in each state (q, s), each variable x can only be opened if $s(x) = \mathtt{w}$, and only be closed if $s(x) = \mathtt{o}$. The initial state (q_0, s_0) and the final state (q_f, s_f) ensure that every variable is opened and closed exactly once. Finally, as A_{fun} has exactly $3^k|Q|$ states, this proves the upper bound for vset-automata.

Lower bound for vset-automata: Let $\mathtt{a} \in \Sigma$, $k \geq 1$, and $X_k := \{x_1, \dots, x_k\} \subset \Xi$. We define the following vset-automaton A_k with variables X_k:


$\mathtt{a}$,
$\vdash_{x_1}, \dots, \vdash_{x_k}$,
$\dashv_{x_1}, \dots, \dashv_{x_k}$

In the terminology of [13], A_k defines the universal spanner over $\{\mathtt{a}\}$ with variables X_k. 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 $u_s \cdot v_s \in \mathsf{Ref}(A_k)$ for each $s \in S$. Furthermore, $u_s \cdot v_s$ does not contain any subword from Γ^2. Hence, according to Lemma 3.7, $u_s v_s \in \mathsf{Ref}(A)$ must hold for every vset-automaton A with $[\![A]\!] = [\![A_k]\!]$.

Let $A \in \mathsf{VA}_{\mathsf{set}}$ 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 $s \neq t$, we have that $u_s \cdot v_t \notin \mathcal{R}(A)$ 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, the ref-word $u_s \cdot v_t$ cannot contain both of $\vdash_{x_i}$ and $\dashv_{x_i}$ exactly once.

This allows us to use Lemma 3.8: For each $s \in S$, we observe $u_s \cdot v_s \in \mathcal{R}(A)$ and $u_s \cdot v_t \notin \mathcal{R}(A)$ for all $t \in S$ with $t \neq s$. 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: Assume $A = (Q, q_0, q_f, \delta)$ is a vstk-automaton with $k \geq 1$ variables. Our goal is to construct a functional vstk-automaton A_{fun} with $(k+2)2^{k-1}|Q|$ states and $[\![A_{\mathsf{fun}}]\!] = [\![A]\!]$. On a conceptual level, the construction is very similar to the vset-automata construction above. The only difference is what information on the variables is stored in the states. For vstk-automata, we store which variables have been opened (to ensure that every variable is opened exactly once), and how many variables have been closed (to ensure that every variable is closed at least once, and to prevent processing $\dashv$ when no variables can be closed). We now define $A_{\mathsf{fun}} := (Q_{\mathsf{fun}}, q_0^{\mathsf{fun}}, q_f^{\mathsf{fun}}, \delta_{\mathsf{fun}})$ in the following way:

- $Q_{\mathsf{fun}} := \{(q, O, i) \mid q \in Q, O \subseteq \mathsf{SVars}\,(A), 0 \leq i \leq |O|\}$,
- $q_0^{\mathsf{fun}} := (q_0, \emptyset, 0)$,
- $q_f^{\mathsf{fun}} := (q_f, \mathsf{SVars}\,(A), k)$,
- $\delta_{\mathsf{fun}}((p, O, i), a) := \{(q, O, i) \mid q \in \delta(p, a)\}$ for $a \in (\Sigma \cup \{\varepsilon\})$ and $(p, O, i) \in Q_{\mathsf{fun}}$,
- for all $(p, O, i) \in Q_{\mathsf{fun}}$ and all $x \in \mathsf{SVars}\,(A)$, let

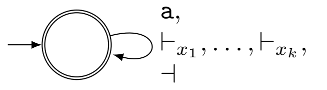
$$\delta_{\mathsf{fun}}((p, O, i), \vdash_x) := \begin{cases} \emptyset & \text{if } x \in O, \\ \{(q, O \cup \{x\}, i) \mid q \in \delta(p, \vdash_x)\} & \text{if } x \notin O, \end{cases}$$

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

It is now easy to see that A_{fun} simulates A. In addition to this, the definition of δ_{fun} ensures that variables are only opened if they have not been opened before (as $\vdash_x$ can only be precessed if $x \notin O$), and that variables can only be closed if there are sufficiently many open variables (as $\dashv$ can only be processed if $i < |O|$). Furthermore, A_{fun} accepts only if every variable has been opened, and if k variables have been closed. Hence, A_{fun} is functional and equivalent to A. All that remains for this upper bound is to prove that $|Q_{\mathsf{fun}}| = (k+2)2^{k-1}|Q|$. First, note that in the definition of Q_{fun}, each state of Q is paired with an element of the set $M := \{(O, i) \mid O \subseteq \mathsf{SVars}\,(A), 0 \leq i \leq |O|\}$. 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, we have $(j+1)$ choices for i. By simplifying this formula (e. g.

using ones favorite software), we obtain $|M| = (k+2)2^{k-1}$. As $|Q_{\mathsf{fun}}| = |M||Q|$, this concludes the proof of the upper bound.

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



In the terminology of [13], A_k defines the universal hierarchical spanner over $\{\mathtt{a}\}$ with variables X_k. Again, we want to define a sequence of pairs of ref-words that allows us to use Lemma 3.8. 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 $u_{O,i} := u_1^O \cdots u_k^O (\dashv\mathtt{a})^i$ and $v_{O,i} := v_1^O \cdots v_k^O (\dashv\mathtt{a})^{k-i}$ by

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

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 3.7 applies, 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 3.8, and conclude that A has at least $|M|$ states. As we established in the proof of the upper bound, $|M| = (k+2)2^{k-1}$. □

We also briefly compare vset- and vstk-automata: It was shown in [13] that $[\![\mathsf{VA}_{\mathsf{stk}}]\!] \subset [\![\mathsf{VA}_{\mathsf{set}}]\!]$. This inclusion is proper for the following reason: As vstk-automata always close the variable that was opened most recently, they can only express hierarchical spanners (a spanner is hierarchical if it contains only w-tuples with non-overlapping spans; for a formal definition, see [13]). While this behavior can be simulated with vset-automata, a slight modification of the proof of Proposition 3.9 shows that this causes an exponential blowup.

Proposition 3.10 *For every $k \geq 1$, there is a vstk-automaton A_k with one state and k variables, such that every vset-automaton A with $[\![A]\!] = [\![A_k]\!]$ has at least $k!$ states.*

Proof. Let $\mathtt{a} \in \Sigma$, $k \geq 1$, and $X_k := \{x_1, \dots, x_k\} \subset \Xi$. We use the same vstk-automaton A_k as in the proof of the lower bound for vstk-automata in Proposition 3.9. We now focus on the following subset of $\mathsf{Ref}(A_k)$:

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

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

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

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 3.7 implies that $R'_k \subseteq \mathcal{R}(A)$. For every permutation $p \in \mathsf{Perm}(k)$, we now define

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

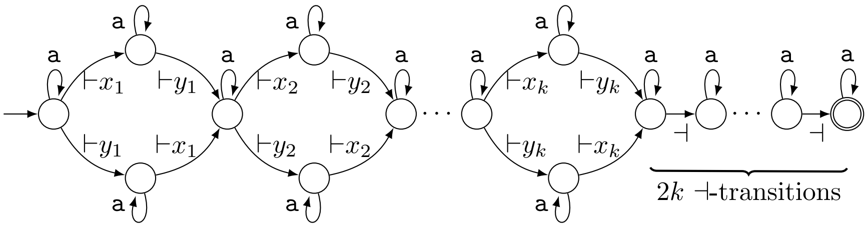
Then $u_p v_p \in R'_k$ holds for every $p \in \mathsf{Perm}(k)$, which implies $u_p v_p \in \mathcal{R}(A)$. Next, consider any $p, q \in \mathsf{Perm}(k)$ with $p \neq q$, and let $r := u_p v_q$. Choose the largest i for which $p(i) \neq q(i)$. As $p(j) = q(j)$ for all $j > i$, the ref-words v_p and v_q have the common prefix $\dashv_{x_{p(k)}} \mathtt{a} \cdots \dashv_{x_{p(i+1)}} \mathtt{a}$, and the leftmost letters where the two ref-words disagree are $\dashv_{x_{p(i)}}$ and $\dashv_{x_{q(i)}}$ (in v_p and in v_q, respectively). In u_p, the variable $x_{p(i)}$ 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, the variable $x_{q(i)}$ 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. By Lemma 3.8, A has at least $|\mathsf{Perm}(k)| = k!$ states. □

To obtain an exponential upper bound, one can construct a vset-automaton that stores a set of variables that have been opened, and a stack of variables that are currently open.

While the proof of Proposition 3.10 uses non-functional vstk-automata, we can observe a lower bound for functional vstk-automata that is not $k!$, but still exponential in k.

Proposition 3.11 *For every $k \geq 1$, there is a functional vstk-automaton A_k with $5k$ states and $2k$ variables, such that every vset-automaton A with $[\![A]\!] = [\![A_k]\!]$ has at least 2^k states.*

Proof. Let $\mathtt{a} \in \Sigma$ and $k \geq 1$. We define the functional vstk-automaton A_k with variables $\{x_1, y_1 \dots, x_k, y_k\} \subset \Xi$ as follows:



Observe that A_k has $5k+1$ states: the starting state, $3k$ states that handle opening the variables, and $2k$ states that handle closing the variables. Intuitively, for each pair x_i and y_i of variables, A_k chooses whether it opens first x_i and then y_i, or vice versa. As there are k such pairs of variables, there are 2^k different combinations of choices.

Building on this, the proof proceeds analogously to that of Proposition 3.10: We can restrict our considerations to those ref-words where exactly one $\mathtt{a}$ is read after each variable operation, which allows us to use Lemma 3.7. For each of the 2^k combinations of choices, we can define a pair (u_i, v_j) of ref-words for vstk-automata, where u_i corresponds to opening the variables and v_i to closing them. We then invoke Lemma 3.8 to conclude that every vset-automaton that is equivalent to A_k needs to have at least 2^k states. □

In contrast to Proposition 3.10, these functional vstk-automata have more than a single state. This is to be expected, as it is easily seen that a functional vstk-automaton with k variables needs to have at least $2k+1$ states (it needs at least $2k$ transitions for the variable operations, each of which has to lead a new state in order to guarantee functionality).

We conclude that although vstk-automata can express less than vset-automata, they may offer an exponential succinctness advantage; and this advantage is orthogonal to the advantage of non-functional over functional automata. We revisit these succinctness issues in Section 4.

4 SpLog: A Logic for Spanners

In this section, we introduce SpLog as a fragment of $\mathsf{EC}^{\mathsf{reg}}$ and connect it to core spanners. Section 4.1 discusses the definitions and the main result; Section 4.2 contains the proof of the main result.

4.1 The Logic

As shown by Freydenberger and Holldack [16], every element of $\mathsf{RGX}^{\mathsf{core}}$ can be converted into an $\mathsf{EC}^{\mathsf{reg}}$-formula, and every word equation with regular constraints can be converted to $\mathsf{RGX}^{\mathsf{core}}$ (and so can every $\mathsf{EC}^{\mathsf{reg}}$-formula; see the comments after Example 2.1). While conversion from word equations or $\mathsf{EC}^{\mathsf{reg}}$ 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. Hence, the spanner can only simulate satisfiability, but not evaluation. Moreover, this construction can lead to an exponential blowup. To overcome these problems, we introduce SpLog (short for *spanner logic*), a fragment of $\mathsf{EC}^{\mathsf{reg}}$ that directly corresponds to core spanners.

Definition 4.1 A formula $\varphi \in \mathsf{EC}$ is *safe* if the following 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 $\mathsf{W} \in \Xi$. Then $\mathsf{SpLog}(\mathsf{W})$, the set of all SpLog-*formulas with main variable* 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 $\varphi(\mathsf{W}; x_1, \ldots, x_k)$ to denote a formula with main variable W, and $\mathsf{free}(\varphi) = \{\mathsf{W}, x_1, \ldots, x_k\}$. To account for the special role of the main variable, we also use $[\![\varphi]\!](w)$ to denote the set of all $\sigma \in [\![\varphi]\!]$ that satisfy $\sigma(\mathsf{W}) = w$.

Definition 4.1 can be seen as restricting the definition of $\mathsf{EC}^{\mathsf{reg}}$. For some purposes, in particular when extending SpLog as we shall do in Section 8, it is more convenient to deal with a recursive definition. Hence, before we consider some example formulas, we introduce the following recursive definition of SpLog, which is equivalent to Definition 4.1.

Definition 4.2 Let $\mathsf{W} \in \Xi$. Then $\mathsf{SpLog}(\mathsf{W})$, the set of all SpLog-*formulas with main variable* W, is the subset of $\mathsf{EC}^{\mathsf{reg}}$ that is obtained from the following recursive rules.

B1. $(\mathsf{W} = \eta_R) \in \mathsf{SpLog}(\mathsf{W})$ for every $\eta_R \in ((\Xi - \{\mathsf{W}\}) \cup \Sigma)^*$.

R1. If $\varphi_1, \varphi_2 \in \mathsf{SpLog}(\mathsf{W})$, then $(\varphi_1 \wedge \varphi_2) \in \mathsf{SpLog}(\mathsf{W})$.
R2. If $\varphi_1, \varphi_2 \in \mathsf{SpLog}(\mathsf{W})$ and $\mathsf{free}(\varphi_1) = \mathsf{free}(\varphi_2)$, then $(\varphi_1 \vee \varphi_2) \in \mathsf{SpLog}(\mathsf{W})$.
R3. If $\varphi \in \mathsf{SpLog}(\mathsf{W})$ and $x \in \mathsf{free}(\varphi) - \{\mathsf{W}\}$, then $(\exists x \colon \varphi) \in \mathsf{SpLog}(\mathsf{W})$.
R4. If $\varphi \in \mathsf{SpLog}(\mathsf{W})$ and $x \in \mathsf{free}(\varphi)$, then $(\varphi \wedge \mathsf{C}_A(x)) \in \mathsf{SpLog}(\mathsf{W})$ for every NFA or regular expression A.

Example 4.3 Define $\varphi_1(\mathsf{W}; x) := \exists y \colon \mathsf{W} = xy \wedge \mathsf{C}_{\Sigma^+}(x)$. Then φ_1 is a $\mathsf{SpLog}(\mathsf{W})$-formula, and $\sigma \models \varphi_1$ if and only if $\sigma(x)$ as a non-empty prefix of $\sigma(\mathsf{W})$.

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 for example $\sigma(\mathsf{W}) = \sigma(x)^2$, then $\sigma \models \varphi_2$, even if $\sigma(y) \not\sqsubseteq \sigma(\mathsf{W})$.

Now define SpLog-formulas

$$\varphi_3(\mathsf{W}; x, y) := \big(\exists x_1, x_2 \colon \mathsf{W} = x_1 x x_2\big) \wedge \big(\exists y_1, y_2 \colon \mathsf{W} = y_1 y y_2\big),$$
$$\varphi_4(\mathsf{W}; x, y) := \exists z_1, z_2, z_3 \colon \big(\mathsf{W} = z_1 x z_2 y z_3 \vee \mathsf{W} = z_1 y z_2 x z_3\big).$$

Then $\sigma \models \varphi_3$ if and only if $\sigma(\mathsf{W})$ contains an occurrence of $\sigma(x)$ and one of $\sigma(y)$; and $\sigma \models \varphi_4$ holds if and only if $\sigma(\mathsf{W})$ contains an occurrence of $\sigma(x)$ and one of $\sigma(y)$ that do not overlap. For example, if $\sigma(\mathsf{W}) = \mathtt{banana}$, $\sigma(x) = \mathtt{ban}$, and $\sigma(y) = \mathtt{nana}$, then $\sigma \models \varphi_3$, but not $\sigma \models \varphi_4$. Next, we define the SpLog-formula

$$\varphi_5(\mathsf{W}; x, y) := \exists z_1, z_2, z_3 \colon \big(\mathsf{W} = z_1 x z_2 y z_3 \land \mathsf{C}_{\alpha_{\geq 5}}(z_2)\big) \lor \big(\mathsf{W} = z_1 y z_2 x z_3 \land \mathsf{C}_{\alpha_{\leq 7}}(z_2)\big),$$

where $\alpha_{\geq 5}$ and $\alpha_{\leq 7}$ are regular expressions with $\mathcal{L}(\alpha_{\geq 5}) = \{w \in \Sigma^* \mid |w| \geq 5\}$ and $\mathcal{L}(\alpha_{\leq 7}) = \{w \in \Sigma^* \mid |w| \leq 7\}$. Then $\sigma \models \varphi_5$ if and only if

1. $\sigma(\mathsf{W})$ contains an occurrence of $\sigma(x)$ to the left of an occurrence of $\sigma(y)$ with at least five terminals between them, or
2. $\sigma(\mathsf{W})$ contains an occurrence of $\sigma(y)$ to the left of an occurrence of $\sigma(x)$ with at most seven terminals between them. ◇

For more examples, see Example 4.5 below and Section 5, which also contains notational shorthands that simplify writing SpLog-formulas.

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 4.4 *Let $\varphi \in \mathsf{SpLog}(\mathsf{W})$, $x \in \mathsf{free}(\varphi) - \{\mathsf{W}\}$, and $\psi \in \mathsf{SpLog}(x)$ such that W does not occur in ψ. We can compute in polynomial time $\chi \in \mathsf{SpLog}(\mathsf{W})$ 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 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 $x = \eta_R$. Then $\chi \equiv (\varphi \land \psi)$ holds. □

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

Example 4.5 First, note that $\sigma(x) \models \psi_1$ holds for the EC-formula $\psi_1(x, y) := \exists u, v \colon \big(x = uv \land y = vu\big)$ if and only if $\sigma(x)$ is a cyclic permutation of y (and vice versa). For example, this holds if $\sigma(x) = \mathtt{owl}$ and $\sigma(y) = \mathtt{low}$, or if $\sigma(x) = \mathtt{headgear}$ and $\sigma(y) = \mathtt{gearhead}$.

Now assume that we want to extend the formula $\varphi_4(\mathsf{W}; x, y)$ from Example 4.3 with the additional requirement that x is a cyclic permutation of y. We could do this directly using the following formula:

$$\begin{aligned}\psi_2(\mathsf{W}; x, y) := \exists z_1, z_2, z_3 \colon &\big(\mathsf{W} = z_1 x z_2 y z_3 \lor \mathsf{W} = z_1 y z_2 x z_3\big)\\ \land \exists u, v \colon &\Big(\big(\exists z_4, z_5 \colon \mathsf{W} = z_4 x z_5 \land \mathsf{W} = z_4 u v z_5\big)\\ &\land \big(\exists z_6, z_7 \colon \mathsf{W} = z_6 y z_7 \land \mathsf{W} = z_6 v u z_7\big)\Big).\end{aligned}$$

Using Lemma 4.4, we can express this using the following simplified notation:

$$\psi_3(\mathsf{W}; x, y) := \exists z_1, z_2, z_3 \colon \\ \big(\mathsf{W} = z_1 x z_2 y z_3 \lor \mathsf{W} = z_1 y z_2 x z_3\big) \land \exists u, v \colon \big(x = uv \land y = vu\big),$$

where we treat x and y as main variables of subformulas. $\diamond$

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 [16].

Definition 4.6 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 $w \in \Sigma^*$, we have $\sigma \in [\![\varphi]\!](w)$ if and only if $\mu \in P(w)$ where, for each $x \in \mathsf{SVars}(P)$ and $[i, j\rangle := \mu(x)$, both $\sigma(x^P) = w_{[1,i\rangle}$ and $\sigma(x^C) = w_{[i,j\rangle}$.

The intuition behind this definition is that every span $[i, j\rangle$ of w is uniquely identified by its content $w_{[i,j\rangle}$, and by $w_{[1,i\rangle}$, the prefix of w that precedes the span. Hence, we represent every variable x of the spanner with 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 4.7 Let $\varphi \in \mathsf{SpLog}(\mathsf{W})$. A spanner P with $\mathsf{SVars}(P) = \mathsf{free}(\varphi) - \{\mathsf{W}\}$ *realizes* φ if, for all $w \in \Sigma^*$, we have $\sigma \in [\![\varphi]\!](w)$ if and only if there exists some $\mu \in P(w)$ with $w_{\mu(x)} = \sigma(x)$ 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 $\mu \in P(w)$ holds for *one* μ with $w_{\mu(x)} = \sigma(x)$ for all $x \in \mathsf{SVars}(P)$, we could require $\mu \in P(w)$ for *all such* μ. But such a spanner can directly be constructed from a spanner P that satisfies Definition 4.7, by joining P with a universal spanner (cf. [13]), and using string equality selections (for our purposes, this does not affect the complexity, as this paper only considers spanners that allow string equality relations – see the proof of Lemma 8.3 for a use of this construction).

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). A *polynomial size conversion* from C_1 to C_2 is an algorithm that, given some $\rho_1 \in C_1$, computes some $\rho_2 \in C_2$ 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 3.1 to obtain a negative result on conversions of v-automata to SpLog.

Lemma 4.8 $\mathsf{P} = \mathsf{NP}$*, if there is a polynomial time conversion from* $\mathsf{VA_{set}}$ *or* $\mathsf{VA_{stk}}$ *to* SpLog*.*

Proof. We show this by reduction from the problem of checking whether $[\![A]\!](\varepsilon) \neq \emptyset$, which is NP-hard according to Lemma 3.1. Let $A \in \mathsf{VA}$, and assume that we can construct in polynomial time a formula $\varphi \in \mathsf{SpLog}(\mathsf{W})$ that realizes A. Then $[\![A]\!](\varepsilon) \neq \emptyset$ holds if and only if there is a substitution $\sigma \in [\![\varphi]\!](\varepsilon)$. As σ maps every variable in φ to a subword of $\sigma(\mathsf{W}) = \varepsilon$, we have $\sigma(x) = \varepsilon$ for all $x \in \mathsf{free}(\varphi)$. The same applies to all variables that are introduced with existential quantifiers. Hence, $\sigma \models \varphi$ if and only if $\sigma_\varepsilon \models \varphi$, where the substitution σ_ε is defined by $\sigma_\varepsilon(x) := \varepsilon$ for all $x \in \Xi$.

Whether this holds can be easily verified by rewriting φ 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 $\mathsf{C}_{A_C}(x)$ is replaced with 1 if $\varepsilon \in \mathcal{L}(A_C)$, and 0 if $\varepsilon \notin \mathcal{L}(A_C)$ (as A_C 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. □

This result is less negative than it might appear at the first glance, as it relies on very specific circumstances. More specifically, it requires a combination of the fact that deciding $[\![A]\!](\varepsilon) \neq \emptyset$ is NP-hard (Lemma 3.1) for non-functional v-automata with the observation that SpLog-formulas can be evaluated trivially on input ε.

We can avoid these circumstances 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 4.9 *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* $\mathsf{VA}^{\mathsf{core}}$ *to* SpLog*.*

Within the framework of spanners realizing SpLog-formulas (and vice versa), this establishes that core spanners and SpLog have the same expressive power. As the proof of this result is quite lengthy, we first discuss some of its implications. The actual proof can then be found in Section 4.2 (note that the conversion from $\mathsf{RGX}^{\mathsf{core}}$ to SpLog was basically proven in [16], only minor modifications are required).

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 [16]. 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. It uses 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. Generally, $P(\varepsilon)$ can be considered a pathological edge case: As $P(w)$ can be understood as search in w, $P(\varepsilon)$ corresponds to a search in the empty word (arguably not a particularly interesting text to search).

But even if we insist on this case, we are still able to observe conversions that might not run in polynomial time, but produce a formula of polynomial size. Furthermore, this is only an issue when dealing with non-functional v-automata; for functional v-automate, we can handle this special case in polynomial time.

Corollary 4.10 *There are polynomial size conversions from* $\mathsf{VA}^{\mathsf{core}}$ *to* SpLog*. These conversions run in polynomial time if all v-automata in the spanner representation are functional.*

Proof. The polynomial time conversions modulo ε from Theorem 4.9 also imply a 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 ε, 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.

If we consider only functional v-automata, Lemma 3.6 ensures that this question be decided in polynomial time, which makes the conversion a polynomial time conversion. □

As discussed in Section 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 4.9 is that these blowups disappear when we can also use the core-algebra.

Corollary 4.11 *Given* $\rho \in \mathsf{VA}^{\mathsf{core}}$*, we can compute an equivalent* $\rho_f \in \mathsf{VA}^{\mathsf{core}}_{\mathsf{set}}$ *or* $\rho_f \in \mathsf{VA}^{\mathsf{core}}_{\mathsf{stk}}$*, where*

1. *ρ_f is of polynomial size,*
2. *every v-automaton in ρ_f is functional,*
3. *every join $\bowtie$ in ρ_f is a cross product $\times$.*

Proof. First, note that the proof of Theorem 4.9 constructs spanner representations that use $\times$ instead of $\bowtie$, and that the constructed v-automata are functional. Hence, we can take a spanner representation $\rho \in \mathsf{VA}^{\mathsf{core}}$, and convert it into a SpLog-formula φ, which is then converted into a spanner representation $\hat{\rho} \in \mathsf{VA}_{\mathsf{set}}$ or $\hat{\rho} \in \mathsf{VA}_{\mathsf{stk}}$. We need one additional step, as the conversion to φ doubles the number of variables (as every x is turned into an x^P and an x^C). In order to obtain ρ_f, we join $\hat{\rho}$ with $x^P\{\Sigma^*\} \cdot x^C\{\Sigma^*\} \cdot \Sigma^*$ for every x, and then project away the x^P. It is also possible to solve this with $\times$ instead of $\bowtie$: For every x, we define a spanner $x_N\{\Sigma^*\} \cdot x\{\Sigma^*\} \cdot \Sigma^*$ (where x_N is a new variable), which we combine with $\hat{\rho}$ by use of $\times$. Before projecting the variables x_N, x^P, and x^C away, we select $\zeta^{=}_{x_N, x^P}$ and $\zeta^{=}_{x, x^C}$. □

Again, if non-functional automata are involved, Lemma 3.1 ensures that computing an equivalent representation ρ_f in polynomial time would imply $\mathsf{P} = \mathsf{NP}$; but we can compute in polynomial time a representation ρ_f that is equivalent modulo ε. On the other hand, if all automata in ρ are functional, we can compute an equivalent representations in polynomial time without relaxing the requirements to modulo ε.

Corollary 4.11 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 4.9, 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 6.2). But we observe 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 4.12 *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.*

Proof. We begin with the combined complexity: NP-hardness follows from the NP-hardness of evaluation of $\mathsf{RGX}^{\mathsf{core}}$, as shown in [16] (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 [16] 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 σ, it suffices to guess a substitution for every variable that is existentially quantified in φ, 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 of φ 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 φ. $\square$

Theorem 4.9 also shows that the PSPACE upper bounds of deciding satisfiability and hierarchicality for $\mathsf{RGX}^{\mathsf{core}}$ that were observed in [16] 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.

Finally, the undecidability results of for core spanners from [16] also carry over to SpLog. This means universality, containment, and equivalence are undecidable; and that adding negation turns SpLog into an undecidable theory. There are also effects on the relative succinctness of SpLog-formulas (see Section 4.2 in [16]). We briefly discuss aspects of this in Section 7.4.

4.2 Proof of Theorem 4.9

Due to its length, we split the proof of Theorem 4.9 into multiple sections. The conversions from SpLog to spanner representations can be found in Section 4.2.1, while the conversions from spanner representations to SpLog are distributed over Section 4.2.2 to 4.2.5 as follows:

1. First, we consider the conversion of primitive spanner representations:
   (a) For regex formulas, see Section 4.2.2.
   (b) For vset-automata, see Section 4.2.3.
   (c) For vstk-automata, see Section 4.2.4.
2. We then examine the conversion of spanner operators in Section 4.2.5.

Two parts of the conversion to SpLog were already shown in [16]: The conversion of regex formulas to $\mathsf{EC}^{\mathsf{reg}}$ from [16] only requires a minimal modification that ensures safety (see Section 4.2.2), and the construction for spanner operators can be used directly (see Section 4.2.5). We repeat these constructions below in order to present all parts of the conversion procedure in one place, and to show that these constructions really result in SpLog-formulas.

4.2.1 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.

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 ε (or an equivalent functional automaton).

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

Clearly, ρ can be computed in polynomial time. Note that the regex formula α 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 α into a functional vset- or vstk-automaton. Furthermore, the projection π_Y ensures that $\mathsf{SVars}(\rho) = \mathsf{var}(\eta_R) = \mathsf{free}(\eta) - \{\mathsf{W}\}$.

In order to see that the construction is correct, first assume that there is a substitution σ 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 want to construct $\mu \in [\![\rho]\!](w)$ with $w_{\mu(x_i)} = w_i$ for $1 \leq i \leq k$. To this end, consider the ref-word $r = r_1 \cdots r_n$, where each r_i is defined as follows: If $\eta_i \in \Sigma$, then $r_i := \eta_i$. Else, we have

$\eta_i = x$ for some $x \in \Xi$. Now, if $i = l_x$, then $r_i := \vdash_x \sigma(x) \dashv_x$. Otherwise, let $r_i := \vdash_{x^{(i)}} \sigma(x) \dashv_{x^{(i)}}$. As $\sigma(\mathsf{W}) = \sigma(\eta_R) = w$, we know that $\mathsf{clr}(r) = w$. Hence, and as r follows the same construction principle as α, we observe $r \in \mathsf{Ref}(\alpha, w)$. Furthermore, $w_{\mu^r(x)} = w_{\mu^r(x^{(i)})} = \sigma(x)$ holds for all $x \in \mathsf{var}(\eta)$ and all $i > \ell_x$ with $\eta_i = x$. Thus, $\mu^r \in [\![S\alpha]\!](w)$. 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 α 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 $i = \ell_x$,
3. $r_i = \vdash_{x^{(i)}} u_i \dashv_x$, with $u_i \in \Sigma^*$, $x \in \Xi$, and $i > \ell_x$.

Furthermore, as $\mu \in [\![S\alpha]\!](w)$, we observe $u_{\ell_x} = u_i$ for all $x \in \Xi$ and all $i > \ell_x$ with $\eta_i = x$. Hence, if we define a substitution σ by $\sigma(\mathsf{W}) := w$, and $\sigma(x) := u_{\ell_x}$ 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 $\psi := (\varphi \wedge \mathsf{C}_A(x))$. Recall that SpLog-formulas are safe; hence, constraint symbols occur only as part of formulas $\varphi \wedge \mathsf{C}_A(x)$, with $x \in \mathsf{free}(\varphi)$. Let ρ_φ be an appropriate spanner representation that realizes φ, and let x^T be a new variable. If A is a regular expression and our goal is to construct a regex formula, let $\rho_A := \Sigma^* \cdot x_T\{A\} \cdot \Sigma^*$. Likewise, if A is an NFA, we can directly construct a corresponding v-automaton ρ_A. 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 ρ realizes ψ, observe that for all w, we have that $\mu \in [\![\rho]\!](w)$ 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 ρ_1 and ρ_2. As ψ 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 ρ realizes ψ 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 ρ_1 and ρ_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 ρ_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 $x \in Y$. Note that this is indeed $\times$ (instead of a more general $\bowtie$), as the renaming ensures that ρ_1 and $\hat{\rho}_2$ have no common variables. 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 some $\hat{\mu}_2 \in [\![\hat{\rho}_2]\!](w)$ such that $w_{\mu(x)} = w_{\hat{\mu}_2(x^T)}$ for all $x \in Y$. Define μ_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 ρ realizes ψ.

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

4.2.2 Conversion of Functional Regex Formulas

As mentioned above, the construction in this section was already presented by Freydenberger and Holldack in the proof of Theorem 3.12 in [16]. Although that proof constructs some $\mathsf{EC}^{\mathsf{reg}}$-formulas that are not SpLog-formulas in a strict sense, Lemma 4.4 allows us to interpret these cases as SpLog-formulas (we shall mention where this is relevant).

Consider a functional regex formula $\rho \in \mathsf{RGX}$. Our goal is to construct a formula $\varphi_\rho \in \mathsf{SpLog}_{\mathsf{rx}}(\mathsf{W})$ that realizes ρ. As explained in [16], we can assume that ρ does not contain $\emptyset$, by rewriting ρ in polynomial time if necessary[4].

Throughout the construction, we use $\vec{x}_{[i..j]}$ as shorthand notation for $x_i^P, x_i^C \ldots, x_j^P, x_j^C$ (with $\vec{z}_{[i..j]}$ defined analogously). We now distinguish the following cases:

1. If ρ does not contain any variables, the ρ is a regular expression, and we define $\varphi_\rho(\mathsf{W}) := \exists x\colon (\mathsf{W} = x \wedge \mathsf{C}_\rho(x))$.
2. If ρ contains variables, we assume that $\mathsf{SVars}(\rho) = \{x_1, \ldots, x_k\}$ with $k \geq 1$. As ρ is functional by definition of RGX, no variable of ρ may occur inside of a Kleene star. Hence, we can distinguish three cases:
   (a) $\rho = \rho_1 \vee \rho_2$, where $\rho_1, \rho_2 \in \mathsf{RGX}$ with $\mathsf{SVars}(\rho_1) = \mathsf{SVars}(\rho_2) = \mathsf{SVars}(\rho)$. We define
   $$\varphi_\rho(\mathsf{W}; \vec{x}_{[1..k]}) := \big(\varphi_{\rho_1}(\mathsf{W}; \vec{x}_{[1..k]}) \vee \varphi_{\rho_1}(\mathsf{W}; \vec{x}_{[1..k]})\big).$$
   (b) $\rho = \rho_1 \cdot \rho_2$, where $\rho_1, \rho_2 \in \mathsf{RGX}$ 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 assume $\mathsf{SVars}(\rho_1) = \{x_1, \ldots, x_m\}$ and $\mathsf{SVars}(\rho_2) = \{x_{m+1}, \ldots, x_k\}$ with $0 \leq m \leq k$. We define
   $$\begin{aligned}\varphi_\rho(\mathsf{W}; \vec{x}_{[1..k]}) := \exists y_1, y_2, \vec{z}_{[m+1..k]}\colon \\ \Big((\mathsf{W} = y_1 \cdot y_2) \wedge \varphi_{\rho_1}(y_1; \vec{x}_{[1..m]}) \wedge \varphi_{\rho_2}(y_2; \vec{z}_{[m+1..k]}) \\ \wedge \bigwedge_{m+1 \leq i \leq n} \big((x_i^P = y_1 \cdot z_i^P) \wedge (x_i^C = z_i^C)\big)\Big).\end{aligned}$$
   Note that Lemma 4.4 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).

[4] The rewriting rules for this are 1. $\emptyset^* \to \varepsilon$, 2. $(\hat\alpha \vee \emptyset) \to \hat\alpha$ and $(\emptyset \vee \hat\alpha) \to \hat\alpha$, 3. $(\hat\alpha \cdot \emptyset) \to \emptyset$ and $(\emptyset \cdot \hat\alpha) \to \emptyset$, and 4. $x\{\emptyset\} \to \emptyset$.

(c) $\rho = x\{\hat{\rho}\}$ for some $x \in \{x_1, \dots, x_k\}$, 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

$$\varphi_\rho(\mathsf{W}; \vec{x}_{[1..k]}) := \Big((x_1^P = \varepsilon) \wedge (\mathsf{W} = x_1^C) \wedge \varphi_{\hat{\rho}}(\mathsf{W}; \vec{x}_{[2..k]})\Big).$$

This case also uses Lemma 4.4.

Clearly, the size of φ_ρ is polynomial in the size of ρ. Furthermore, we construct φ_ρ by following the syntax of ρ without any expensive additional computations. Therefore, we conclude that φ_ρ can be computed in polynomial time. For the proof of correctness and further explanations, see Theorem 3.12 in [16].

4.2.3 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 [13], this is ensured in the definition of accepting runs; our ref-word definition ensures this through Ref).

While one could encode all possible combinations of how variables overlap, this would result in a formula with a size that is exponential in the number of variables. 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 variable operations (opening or closing). Furthermore, there exist states $s_0, \dots, s_{2k}, t_0, \dots, t_{2k} \in Q$ such that the following holds:

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

In other words, each s_i is the state between processing v_i and w_i, and t_i is the state between w_i and v_{i+1}. Also see Figure 3.

The main idea is that special variables represent all states s_i and t_i, and in which v_i 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, we can work around this: For each piece of information that is represented (e. g., for each state s_i), we define a group of variables that represents the possible choices (e. g. variables s_i^q for all $q \in Q$), and ensure

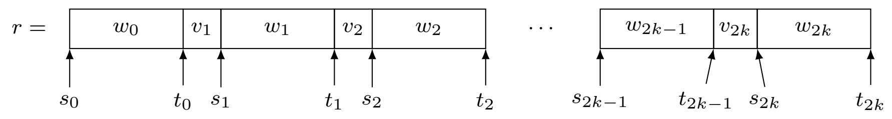


Fig. 3 A graphical representation of the factorization of $r \in \mathsf{Ref}(A, w)$, which is used in the proof of Theorem 4.9, Section 4.2.3. The s_i and t_i denote the states before and after processing the w_i, while the v_i denote variable operations.

that for every satisfying σ, exactly one of these variables is mapped to the first letter of W, while all others are mapped to ε.

Our goal is constructing a $\mathsf{SpLog}(\mathsf{W})$-formula φ that realizes A on all $w \in \Sigma^+$ (the case of $w = \varepsilon$ is ignored, as the conversion works modulo ε). Assuming $w \neq \varepsilon$ allows us to define the formula $\varphi_{\hat{a}} := \exists \hat{w}\colon (\mathsf{W} = \hat{a} \cdot \hat{w})$, which stores the first letter of w in $\hat{a}$, the special variable that we shall use to synchronize various subformulas.

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 ε. This allows us to synchronize different parts of φ, and to store non-deterministic decisions, like the assigned states. The sets are as follows:

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

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

On an intuitive level, $\varphi_{s,i}^q$ represents that $s_i = q$ (likewise, $\varphi_{t,i}^q$ represents $t_i = q$). 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 4.4, 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 only if we assume that $\hat{a}$ refers to the first letter of W, which shall be ensured by $\varphi_{\hat{a}}$. Further down, the fact that the set of free variables of $\varphi_{s,i}^q$ depends only on i, and not on q, shall allow us to use these formulas in disjunctions.

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

$$\varphi^j_{o,i} := (o^j_i = \hat{a}) \wedge \bigwedge_{\substack{1\leq l\leq 2k,\\ l\neq j}} (o^l_i = \varepsilon), \qquad \varphi^j_{c,i} := (c^j_i = \hat{a}) \wedge \bigwedge_{\substack{1\leq l\leq 2k,\\ l\neq j}} (c^l_i = \varepsilon),$$

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

While the formulas $\varphi^j_{o,i}$ and $\varphi^j_{c,i}$ 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

$$\varphi^{\vdash,j}_{v,i} := \big((o^i_j = \hat{a}) \wedge (c^i_j = \varepsilon)\big) \wedge \bigwedge_{\substack{1\leq l\leq k,\\ l\neq j}} \big((o^i_l = \varepsilon) \wedge (c^i_l = \varepsilon)\big)\,,$$

$$\varphi^{\dashv,j}_{v,i} := \big((o^i_j = \varepsilon) \wedge (c^i_j = \hat{a})\big) \wedge \bigwedge_{\substack{1\leq l\leq k,\\ l\neq j}} \big((o^i_l = \varepsilon) \wedge (c^i_l = \varepsilon)\big)\,.$$

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

$$\varphi := \exists \vec{v}\colon \varphi_{\hat{a}} \wedge \varphi_{\mathsf{fact}} \wedge \varphi_{\mathsf{init}} \wedge \varphi_{\mathsf{final}} \wedge \varphi_{\mathsf{span}} \wedge \varphi_{\mathsf{t-trans}} \wedge \varphi_{\mathsf{v-trans}},$$

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 φ are defined as follows:

- $\varphi_{\mathsf{fact}} := (\mathsf{W} = w_0 \cdot w_1 \cdots w_{2k})$. This factorizes w into $w = w_0 \cdot w_1 \cdots w_{2k}$.
- $\varphi_{\mathsf{init}} := \varphi^{q_0}_{s,0}$. This ensures $s_0 = q_0$
- $\varphi_{\mathsf{final}} := \varphi^{q_f}_{t,2k}$. This expresses $t_{2k} = q_f$.
- φ_{span} is defined as

$$\bigwedge_{i=1}^{k} \bigvee_{j=1}^{2k-1} \bigvee_{l=j+1}^{2k} \big(\varphi^j_{o,i} \wedge \varphi^l_{c,i} \wedge \varphi_{\mathsf{fact}} \wedge \big(x^P_i = w_0 \cdots w_{j-1}\big) \wedge \big(x^C_i = w_j \cdots w_{l-1}\big)\big)$$

  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 φ_{span} is safe, note that or each i, the formula consists of a disjunction of formulas, each of which has the free variables

$$O_i \cup C_i \cup \{\mathsf{W}, \hat{a}, w_0, \ldots, w_{2k}, x^P_i, x^C_i\}.$$

– $\varphi_{\mathsf{t-trans}}$ covers terminal transitions. It ensures that each w_i corresponds to a path from s_i to t_i in A, where the transitions along the path have labels from $\Sigma \cup \{\varepsilon\}$. In order to define this, for each pair $p, q \in Q$, we define an NFA $A_{p,q} := (Q, p, q, \delta_{p,q})$, where $\delta_{p,q}$ is the restriction of δ to $Q \times (\Sigma \cup \varepsilon) \to 2^Q$. In other words, for all $\hat{q} \in Q$ and all $\lambda \in (\Sigma \cup \{\varepsilon\} \cup \Gamma)$,

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

Hence, each $A_{p,q}$ is the NFA over Σ that simulates A when starting in p, accepting in q, and using no variable transitions. Then we define

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

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, $\varphi_{\mathsf{t-trans}}$ states that for each $0 \leq i \leq 2k$, $w_i \in \mathcal{L}(A_{s_i,t_i})$, which is equivalent to $t_i \in \delta(s_i, w_i)$.

– $\varphi_{\mathsf{v-trans}}$ covers variable transitions. It ensures $s_i \in \delta(t_{i-1}, v_i)$ for each v_i. We define $\varphi_{\mathsf{v-trans}}$ as

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

Here, $\varphi_{\mathsf{v-trans}}$ considers each v_i, finds the (unique) j with $v_i = \{\vdash_{x_i}, \dashv_{x_i}\}$, and ensures $s_i \in \delta(t_{i-1}, \vdash_{x_j})$. To see that $\varphi_{\mathsf{v-trans}}$ is 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 φ_{fact}, we have $\sigma(\mathsf{W}) = \sigma(w_0) \cdots \sigma(w_{2k})$. Furthermore, φ_{span} and $\varphi_{\mathsf{v-trans}}$ ensure that the v_i are valid for a word from $\mathsf{Ref}(A, \sigma(\mathsf{W}))$: Due to $\varphi_{\mathsf{v-trans}}$, every v_i with $1 \leq i \leq 2k$ is assigned exactly one value from the set $\{\vdash_{x_1}, \ldots, \vdash_{x_k}, \dashv_{x_1}, \ldots, \dashv_{x_k}\}$; and due to φ_{span}, for every $1 \leq i \leq k$, there exist exactly one j and one l with $1 \leq j < l \leq k$, 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: φ_{init} and φ_{final} ensure $s_0 = q_0$ and $t_{2k} = q_f$, respectively. For $0 \leq i \leq 2k$, $\varphi_{\mathsf{t-trans}}$ guarantees $t_i \in \delta(q_i, \sigma(w_i))$, while $\varphi_{\mathsf{v-trans}}$ enforces $s_i \in \delta(t_{i-1}, v_i)$ for $1 \leq i \leq 2k$. This allows us to conclude that σ encodes an $r \in \mathsf{Ref}(A, \sigma(\mathsf{W}))$. Finally, φ_{span} also ensures that all span variables x_i^P and 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 substitution $\sigma \in [\![\varphi]\!]$.

Complexity: In oder to prove that φ can be computed in polynomial time, it suffices to show that the size of φ is polynomial in the size of A (as φ is directly derived from the structure of A). Let $n := |Q|$, and recall that $k = |\mathsf{SVars}\,(A)|$. By examining the subformulas, we can determine that φ is of size $O(k^4 + k^2 n^3 + kn^4)$, which is clearly polynomial in the size of A. Hence, φ can be constructed in polynomial time.

4.2.4 Conversion of vstk-Automata

The construction for vstk-automata is very similar to the construction for vset-automata (see Section 4.2.3). 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 4.2.3. This allows us to reuse the construction from the vset-automata case, if we also add a formula φ_{stack} that ensures that variables are closed in the stack order. We define

$$\varphi := \exists \vec{v} : \varphi_{\hat{a}} \wedge \varphi_{\mathsf{fact}} \wedge \varphi_{\mathsf{init}} \wedge \varphi_{\mathsf{final}} \wedge \varphi_{\mathsf{span}} \wedge \varphi_{\mathsf{t-trans}} \wedge \hat{\varphi}_{\mathsf{v-trans}} \wedge \varphi_{\mathsf{stack}},$$

where all formulas are defined as in Section 4.2.3, in addition to the following two new formulas:

- $\hat{\varphi}_{\mathsf{v-trans}}$ is $\varphi_{\mathsf{v-trans}}$, adapted to use $\dashv$ instead of $\dashv_{x_i}$. We define $\hat{\varphi}_{\mathsf{v-trans}}$ as

$$\bigwedge_{i=1}^{2k} \bigvee_{j=1}^{k} \Big(\big(\varphi_{v,i}^{\vdash,j} \wedge \bigvee_{\substack{p \in Q,\\ q \in \delta(p, \vdash_{x_j})}} (\varphi_{t,i-1}^{p} \wedge \varphi_{s,i}^{q})\big) \vee \big(\varphi_{v,i}^{\dashv,j} \wedge \bigvee_{\substack{p \in Q,\\ q \in \delta(p, \dashv)}} (\varphi_{t,i-1}^{p} \wedge \varphi_{s,i}^{q})\big)\Big)$$

  Hence, $\hat{\varphi}_{\mathsf{v-trans}}$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 φ_{stack}).
- φ_{stack} states that each closing operator closes the most recent open variable. To this end, we define φ_{stack} as

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

  In order to understand this formula, let $o_i, c_i \in \{1, \ldots, 2k\}$ 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 φ_{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 φ_{stack}, and from correctness of the construction from Section 4.2.3. Regarding the complexity, we observe that φ_{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 $O(k)$. This leads to a total size of $O(k^7 + k^2 n^3 + kn^4)$, which is larger than for vset-automata, but still polynomial in the size of A.

4.2.5 Putting The Parts Together (Converting Operators)

Here, we can directly use the construction from the proof of Theorem 3.12 in [16]. We use the same shorthand notation $\vec{x}_{[i..j]}$ as in Section 4.2.2. In contrast to Section 4.2.2, we shall use Lemma 4.4 only once.

Consider a representation $\rho \in \mathsf{RGX}^{\mathsf{core}}$ or $\rho \in \mathsf{VA}^{\mathsf{core}}$. To construct a $\mathsf{SpLog}(\mathsf{W})$-formula φ_ρ that realizes ρ, we distinguish the following cases:

1. If ρ is a regex formula or a vset-automaton, we construct φ_ρ as described in the appropriate previous section.
2. $\rho = \pi_Y \hat{\rho}$, with $Y = \mathsf{SVars}\,(\rho)$ and $\mathsf{SVars}\,(\hat{\rho}) \supseteq \mathsf{SVars}\,(\rho)$. Assume w.l.o.g. $Y = \{x_1, \dots, x_n\}$ and $\mathsf{SVars}\,(\hat{\rho}) = \{x_1, \dots, x_{n+m}\}$ with $m, n \geq 0$. We define
$$\varphi_\rho(\mathsf{W}; \vec{x}_{[1..n]}) := \exists \vec{x}_{[n+1..n+m]} \colon \varphi_{\hat{\rho}}\left(\mathsf{W}; \vec{x}_{[1..n+m]}\right).$$
3. $\rho = \zeta^{=}_{\vec{x}} \hat{\rho}$, with $\mathsf{SVars}\,(\rho) = \{x_1, \dots, x_k\}$ where $k \geq 2$, as well as $\vec{x} \in (\mathsf{SVars}\,(\rho))^m$ with $2 \leq m \leq k$, and $\mathsf{SVars}\,(\hat{\rho}) = \mathsf{SVars}\,(\rho)$. Assume w.l.o.g. that $\vec{x} = x_1, \dots, x_k$. We define
$$\varphi_\rho(\mathsf{W}; \vec{x}_{[1..k]}) := \Big(\varphi_{\hat{\rho}}(\mathsf{W}; \vec{x}_{[1..k]}) \land \bigwedge_{2 \leq i \leq k} (x_1^C = x_i^C)\Big).$$
In this case, we use Lemma 4.4 to interpret this as a $\mathsf{SpLog}(\mathsf{W})$-formula.
4. $\rho = (\rho_1 \cup \rho_2)$, with $\mathsf{SVars}\,(\rho_1) = \mathsf{SVars}\,(\rho_2) = \mathsf{SVars}\,(\rho) = \{x_1, \dots, x_k\}$. Let
$$\varphi_\rho(\mathsf{W}; \vec{x}_{[1..k]}) := \left(\varphi_{\rho_1}\left(\mathsf{W}; \vec{x}_{[1..k]}\right) \lor \varphi_{\rho_2}(\mathsf{W}; \vec{x}_{[1..k]})\right).$$
5. $\rho = (\rho_1 \bowtie \rho_2)$ with $\mathsf{SVars}\,(\rho) = \mathsf{SVars}\,(\rho_1) \cup \mathsf{SVars}\,(\rho_2)$. We assume without loss of generality that $\mathsf{SVars}\,(\rho_1) = \{x_1, \dots, x_l\}$ and $\mathsf{SVars}\,(\rho_2) = \{x_m, \dots, x_n\}$ with $0 \leq l \leq n$ and $1 \leq m \leq n+1$. We define
$$\varphi_\rho(\mathsf{W}; \vec{x}_{[1..n]}) := \left(\varphi_{\rho_1}(\mathsf{W}; \vec{x}_{[1..l]}) \land \varphi_{\rho_2}(\mathsf{W}; \vec{x}_{[m..n]})\right).$$

Explanations and a correctness proof can be found in the proof of Theorem 3.12 in [16]. As φ_ρ can be constructed in polynomial time, this concludes the proof.

5 Expressing Languages and Relations in SpLog

This section examines expressing relations and languages in SpLog: Section 5.1 lays the formal groundwork by introducing selectability of relations in SpLog. Section 5.2 defines a normal form with an example application. Section 5.3 provides an efficient conversion of a subclass of xregex to SpLog.

5.1 Selectable Relations

One of the topics of Fagin et al. [13] 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, [13] 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 $\vec{x} = (x_1, \dots, x_k)$ of variables with $x_1, \dots, x_k \in \mathsf{SVars}(\rho)$, the spanner $[\![\zeta^R_{\vec{x}}\rho]\!]$ is expressible in $\mathsf{RGX}^{\mathsf{core}}$, where ζ^R is the generalization of $\zeta^=$ to R. More specifically, $[\![\zeta^R_{\vec{x}}\rho]\!](w)$ is defined as the set of all $\mu \in [\![\rho]\!](w)$ for which $\big(w_{\mu(x_1)}, \dots, w_{\mu(x_k)}\big) \in R$.

Analogously, we say that R is *SpLog-selectable* if for every $\varphi \in \mathsf{SpLog}(\mathsf{W})$ and every sequence $\vec{x} = (x_1, \dots, x_k)$ of variables with $x_1, \dots, x_k \in \mathsf{free}(\varphi) - \{\mathsf{W}\}$, there is a SpLog-formula $\varphi^R_{\vec{x}}$ 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 5.1 *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 $\varphi(W; x_1, \dots, x_k) \in \mathsf{SpLog}$ such that for all σ, we have $\sigma \models (\exists \mathsf{W}\colon \varphi)$ if and only if $(\sigma(x_1), \dots, \sigma(x_k)) \in R$.*

Proof. Before we begin with the proof, not that we include the existential quantifier $\exists\mathsf{W}$ in the third condition to make the notation more elegant. Without this, we would need to require $\sigma(x_i) \sqsubseteq \sigma(\mathsf{W})$ for all x_i in addition to $(\sigma(x_1), \dots, \sigma(x_k)) \in R$ to make the "if"-direction of the condition work. Choose $R \subseteq (\Sigma^*)^k$, $k \geq 1$.

Equivalence of conditions 1 and 2: We first prove that R 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 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 constructing a formula φ^R such that $\sigma \models \varphi^R$ if and only if $\sigma \models \varphi$ and $(\sigma(x_1), \dots, \sigma(x_k)) \in R$. According to Theorem 4.9, there exists a representation $\rho \in \mathsf{RGX}^{\mathsf{core}}$ that realizes φ. More explicitly, this means that $\mathsf{SVars}(\rho) = \mathsf{free}(\varphi) - \{\mathsf{W}\}$, and for every $w \in \Sigma^*$, we have $\sigma \in [\![\varphi]\!](w)$ if and only if there exists some $\mu \in [\![\rho]\!](w)$ with $w_{\mu(x)} = \sigma(x)$ for all $x \in \mathsf{SVars}(\rho)$.

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

Again by Theorem 4.9, there exists a formula $\hat{\varphi}^R \in \mathsf{SpLog}$ that realizes ρ^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 $w \in \Sigma^*$, we have $\sigma \in [\![\varphi^R]\!](w)$ if and only if there exists some $\mu \in [\![\rho^R]\!](w)$ with $w_{\mu(x)} = \sigma(x)$ 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 4.9 twice.

Equivalence of conditions 2 and 3: For the "if"-direction, let $\varphi(\mathsf{W}; x_1, \dots, x_k) \in \mathsf{SpLog}(\mathsf{W})$ such that $\sigma \models (\exists \mathsf{W} \colon \varphi)$ if and only if $(\sigma(x_1), \dots, \sigma(x_k)) \in R$. Now, for $\psi \in \mathsf{SpLog}(\mathsf{W})$ and $\vec{x} := (x_1, \dots, x_k) \in (\mathsf{free}(\psi))^k$, 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.

For the "only if"-direction, assume R is SpLog-selectable. We define a $\mathsf{SpLog}(\mathsf{W})$-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 $\varphi \in \mathsf{SpLog}$ such that $\sigma \models \varphi$ if and only if $\sigma \models \psi$ and $(\sigma(x_1), \dots \sigma(x_k)) \in R$. Hence, $\sigma \models (\exists \mathsf{W} \colon \varphi)$ 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 4.9, this does not even affect efficiency.

Before we discuss how the equivalent third condition in Lemma 5.1 can be used to simplify this even further, we consider a short example. As shown by Fagin et al. [13], the relation $\sqsubseteq$ is selectable by core spanners. We reprove this by showing that it is SpLog-selectable.

Example 5.2 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 $y_1 x y_2 = y$. $\diamond$

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 and z_2 in Example 5.2), and, secondly, the basic form that equations have the main variable W on the left side. Together with Lemma 4.4 and the third condition of Lemma 5.1, 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.

Example 5.3 Using the aforementioned simplifications, we can write the formula from Example 5.2 as $\varphi_{\sqsubseteq}(\mathsf{W}; x, y) := \exists y_1, y_2\colon (y = y_1 \cdot x \cdot y_2)$. Similarly, we can select the prefix relation with the formula $\varphi_{\mathsf{pref}}(\mathsf{W}; x, y) := \exists z\colon y = xz$. Both are shorthands for $\mathsf{SpLog}(\mathsf{W})$-formulas.

As mentioned above, this allows us use $x \sqsubseteq y$ as syntactic sugar. Other extensions are $x \neq \varepsilon$ and $x \neq y$: For $x \neq \varepsilon$, we can choose

$$\varphi_{\neq\varepsilon}(\mathsf{W}; x) := (x \sqsubseteq \mathsf{W}) \wedge (\mathsf{C}_{\Sigma^+}(x)).$$

The more general $x \neq y$ is expressed as follows:

$$\begin{aligned}\varphi_{\neq}(\mathsf{W}; x, y) :=& \Big(\big(\exists x_2\colon (x = y x_2) \wedge (x_2 \neq \varepsilon)\big) \vee \big(\exists y_2\colon (y = x y_2) \wedge (y_2 \neq \varepsilon)\big)\Big) \\ &\vee \Big(\bigvee_{a \in \Sigma} \big(\exists z, x_2, y_2, b\colon (x = z a x_2) \wedge (y = z b y_2) \wedge \mathsf{C}_{\Sigma - \{a\}}(b)\big)\Big)\end{aligned}$$

The core spanner selectability of $\neq$ was already shown in [13], 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 [30] to show EC-expressibility of $\neq$, as Σ^+ and $\Sigma - \{a\}$ can be expressed without using constrains; for example by defining $\varphi_{\neq\varepsilon}(\mathsf{W}; x) := \bigvee_{a \in \Sigma}(\exists y\colon x = ay)$. $\diamond$

Example 5.4 In this example, we show that 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 [37], 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 $u_1, u_2 \in \Sigma^*$ and $a, b \in \Sigma$. 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 $d_{\mathsf{L}}(u, v) \leq k$ is SpLog-selectable:

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

where $\alpha := \beta := (\Sigma \vee \varepsilon)$. An insertion is expressed by assigning $x_i = \varepsilon$ and $y_i \in \Sigma$, a deletion 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 handle if less than k operations are used.

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

5.2 A Normal Form for SpLog

Another advantage of using a logic is the existence of normal forms[5], although this should not be misunderstood as a claim that core spanners do not have normal forms. The *core-simplification lemma* (Lemma 4.16 in Fagin et al. [13]) states that every core spanner can be expressed as $\pi_V S A$, where $A \in \mathsf{VA}_{\mathsf{set}}$, $V \subseteq \mathsf{SVars}(A)$, and S is a sequence of selections $\zeta^{=}_{x,y}$ for $x, y \in \mathsf{SVars}(A)$. But as the construction from the proof of Theorem 4.9 converts vset-automata into

[5] "Normal form" in the sense that every formula can be rewritten into an equivalent formula that uses a restricted syntax.

rather complicated formulas, this does not directly translate into a compact normal form for SpLog. Instead, we consider the following normal form, which allows us to study a closure property of the class of SpLog-definable languages (Lemma 5.7 below). We shall also use this normal form in Section 7.3 to establish connections between SpLog and certain types of graph queries.

Definition 5.5 A $\varphi \in \mathsf{SpLog}$ is a *prenex conjunction* if it is of the form $\varphi = \exists x_1, \ldots, x_k \colon (\bigwedge_{i=1}^{m} \eta_i \wedge \bigwedge_{j=1}^{n} C_j)$, with $k, n \geq 0$, $m \geq 1$, where the η_i are word equations, and the C_j are constraints. A SpLog-formula is in *DPC-normal form* if it is a disjunction of prenex conjunctions. Let DPC and PC denote the class of all SpLog-formulas in DPC-normal form and the class of all prenex conjunctions, respectively. We use $\mathsf{DPC}_{\mathsf{rx}}$ and $\mathsf{PC}_{\mathsf{rx}}$ for the subclasses of $\mathsf{SpLog}_{\mathsf{rx}}$.

Lemma 5.6 *Given $\varphi \in \mathsf{SpLog}$, we can compute $\psi \in \mathsf{DPC}$ with $\varphi \equiv \psi$.*

Proof. First, we ensure that for every subformula of φ that has the form $\exists x \colon \psi$, x does not appear in φ outside of ψ. 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 variables. The DPC-normal form can then be computed by applying the following rewriting rules:

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

where $x \in \Xi$, $\varphi_1, \varphi_2 \in \mathsf{SpLog}$, and φ_C is a SpLog-formula or a constraint. These rules are also applied modulo commutation of $\wedge$ and $\vee$; in other words, $\varphi_C \wedge (\varphi_1 \vee \varphi_2)$ 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 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 φ_C is some constraint $C_A(x)$, then $x \in \mathsf{free}(\varphi_1 \vee \varphi_2)$ 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. □

The construction from the proof of Lemma 5.6 might result in an exponential blowup; the author conjectures that this blowup cannot be avoided.

We use DPC-normal form to illustrate some differences between SpLog and $\mathsf{EC}^{\mathsf{reg}}$. First, we define the notion of the language of a formula (in Section 6.1, we shall see that this has applications beyond the language theoretic point of view). Every $\mathsf{EC}^{\mathsf{reg}}$-formula φ defines a language $\mathcal{L}_x(\varphi) := \{\sigma(x) \mid \sigma \models \varphi\}$ for every variable $x \in \mathsf{free}(\varphi)$. If φ has exactly one free variable (say x), we define $\mathcal{L}(\varphi) := \mathcal{L}_x(\varphi)$. For $\mathcal{C} \subseteq \mathsf{EC}^{\mathsf{reg}}$, a language $L \subseteq \Sigma^*$ is a $\mathcal{C}$-language if there is a formula $\varphi \in \mathcal{C}$ with $\mathcal{L}(\varphi) = L$. We denote this by $L \in \mathcal{L}(\mathcal{C})$. Hence, SpLog-languages are always defined by the main variable.

For $L \subseteq \Sigma^*$ and $a \in \Sigma$, we define L/a, the *right quotient of L by a*, as the language of all w with $wa \in L$. The class of $\mathsf{EC}^{\mathsf{reg}}$-languages is closed under this operation, as we have $\mathcal{L}(\varphi_{/a}) = \mathcal{L}(\varphi)/a$ for $\varphi_{/a}(w) := \exists u\colon ((u = wa) \land \varphi(u))$. But as SpLog-variables can only contain subwords of the main variable, writing $u = \mathsf{W}a$ is not possible in $\mathsf{SpLog}(\mathsf{W})$. Hence, our proof for the SpLog-case is more involved and relies on Lemma 5.6.

Lemma 5.7 *$L/a \in \mathcal{L}(\mathsf{SpLog})$ for all $L \in \mathcal{L}(\mathsf{SpLog})$ and all $a \in \Sigma$.*

Proof. Let $\varphi(\mathsf{W}) \in \mathsf{SpLog}(\mathsf{W})$, and let $a \in \Sigma$. It suffices to prove the claim for $\varphi \in \mathsf{PC}$: Assume that φ is not a prenex conjunction. According to Lemma 5.6, $\varphi \equiv \bigvee \varphi_i$ for some $\varphi_i \in \mathsf{PC}$. Hence, $\mathcal{L}(\varphi)/a = \bigcup(\mathcal{L}(\varphi_i)/a)$.

Thus, assume without loss of generality that φ is a prenex conjunction

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

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

Our goal is to bring the α_i into a form where we can easily split off a at the right side. Hence, we consider all possibilities which variables or terminals generate the rightmost letter in a word $w \in \mathcal{L}(\varphi)$. As some variables might be erased,this is not always the rightmost variable of an η_i. To this end, 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, π_N erases the variables from $X - N$, and leaves variables from N and terminals unchanged. For each of these N, we now define a formula

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

where $\vec{x}_N$ contains exactly the variables from N. 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. We observe that $\varphi \equiv \bigvee_{\emptyset \subseteq N \subseteq X} \varphi_N$.

The end goal of the construction is finding formulas ψ_N with $\mathcal{L}(\psi_N) = \mathcal{L}(\varphi_N)/a$ for each set N. As intermediate step, we shall construct formulas χ_N with $\mathcal{L}(\chi_N) = \mathcal{L}(\varphi_N) \cap (\Sigma^* \cdot a)$.

As all remaining variables have to be substituted with non-empty words, we know that some φ_N can only generate a word that ends on $\mathtt{a}$ if every variable that is the rightmost symbol of some $\pi_N(\alpha_i)$ is 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 $1 \leq i \leq m$. But we also need adapt the constraints that refer to variables from R_N: For each $1 \leq j \leq n$, there exist an NFA A and a variable $x \in X$ such that $C_j = \mathsf{C}_A(x)$. If $x \notin R_N$, we define $C_j^{/a} := C_j$. On the other hand, if $x \in R_N$, let $C_j^{/a} := \mathsf{C}_{A_{/a}}(x)$, where $A_{/a}$ is an NFA with $\mathcal{L}(A_{/a}) = \mathcal{L}(A)/a$. As the class of regular languages is closed under $/a$ (proving this is a standard exercise), such an $A_{/a}$ always exists (and although $\mathcal{L}(A_{/a}) = \emptyset$ might hold, this simply results in a formula that is not satisfiable). We combine this to

$$\chi_N := \exists \vec{x}_N \colon \Big(\bigwedge_{i=1}^{m} (\mathsf{W} = \beta_{N,i}) \wedge \bigwedge_{j=1}^{n} C_j^{/a} \wedge \bigwedge_{x \in (N - R_N)} (x \neq \varepsilon)\Big).$$

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

Now we are ready for the final step, splitting off the a. Our goal is to define formulas ψ_N with $\mathcal{L}(\psi_N) = \mathcal{L}(\varphi_N)/a$. We distinguish two cases: Firstly, if, for some N, any of the $\beta_{N,i}$ is ε or ends on some terminal from $\Sigma - \{a\}$, we know that $\mathcal{L}(\varphi_N) \cap (\Sigma^* \cdot a) = \emptyset$, which is equivalent to $\mathcal{L}(\varphi_N)/a = \emptyset$. Hence, we can discard this choice of N. To simplify the presentation, we then assume that ψ_N is 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. Thus, for each $1 \leq i \leq m$, there exists a well-defined $\gamma_{N,i}$ with $\gamma_{N,i} = \beta_{N,i} \cdot a$. We define

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

and observe that $\mathcal{L}(\psi_N) = \mathcal{L}(\chi_N)/a = \mathcal{L}(\varphi_N)/a$. All that remains is to combine the formulas into a single formula $\psi := \bigvee_{\emptyset \subset N \subseteq X} \psi_N$. As $\mathsf{free}(\psi_N) = \{\mathsf{W}\}$ for each N, this is indeed a SpLog-formula. By our previous observations, we can state $\mathcal{L}(\psi) = \bigcup \mathcal{L}(\psi_N) = \bigcup \mathcal{L}(\varphi_N)/a = \big(\bigcup \mathcal{L}(\varphi_N)\big)/a = \mathcal{L}(\varphi)/a$. Hence, the class of SpLog-languages is closed under $/a$. □

The same can be observed for the analogously defined left quotient by a. We use Lemma 5.7 twice in Section 6.2.

5.3 Efficient Conversion of vsf-xregex to SpLog

Most modern implementations of regular expressions use a backreference operator that allows the definition of non-regular languages (see e. g. Freydenberger and Schmid [18] for more details). This is formalized in *xregex* (a. k. a. extended regular expressions, regex, or regular expressions with backreferences), 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 xregex is parsed left to right. We examine two short examples of xregex languages; more can be found in [5,14,18,41].

Example 5.8 Let $\alpha := x\{\Sigma^*\} \cdot \&x \cdot \&x$ and $\beta := x\{\mathtt{aa}^+\}(\&x)^+$. Then $\mathcal{L}(\alpha)$ is the language of all www with $w \in \Sigma^*$, while $\mathcal{L}(\beta)$ is the language of all words $\mathtt{a}^n$, such that $n \geq 4$ is not a prime number[6]. $\diamond$

We give a ref-words based definition of xregex semantics in the following section. Readers who are satisfied with the informal semantics are invited to skip to the discussion of the actual result in Section 5.3.2

5.3.1 Xregex Semantics

We define the semantics of xregex using the ref-word approach by Schmid [41] (for a definition with parse trees, cf. Freydenberger and Holldack [16]).

Recall that the syntax of xregex extends that of regex formulas, by adding the case $\&x$ for all $x \in \Xi$ to the recursive definition. We exclude all cases of variable bindings $x\{\alpha\}$ where α contains $\&x$ or some $x\{\beta\}$.

Likewise, the notion of the *ref-language* $\mathcal{L}(\alpha)$ of an xregex α is obtained by adding the rule $\mathcal{R}(\&x) = x$ for all $x \in \Xi$ to the definition for regex formulas.

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 $\vdash_x u_2 \dashv_x$ if u_3 does not contain $\vdash_x$ (hence, 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 (in other words, these default to ε). 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 Σ^* (cf. [18,41] for more information). Finally, we define $\mathcal{L}(\alpha) := \{\mathsf{D}(r) \mid r \in \mathcal{R}(\alpha)\}$.

5.3.2 Converting vsf-xregex

As shown by Fagin et al. [13], core spanners cannot define all xregex languages (e. g., they cannot express $\mathcal{L}(\beta)$ from Example 5.8, see [16]). But Freydenberger

[6] Originally invented by Abigail [1] as a PERL regular expression.

and Holldack [16] identified a core spanner definable subclass of xregex, the *variable-star-free xregex* (short: *vsf-xregex*). A vsf-xregex is an xregex 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 [16] can lead to an exponential blowup. The question whether a more efficient conversion is possible was left open in [16]. Using SpLog, we answer this positively.

Theorem 5.9 *Given a vsf-xregex α, we can compute in polynomial time $\varphi \in \mathsf{SpLog}$ with $\mathcal{L}(\varphi) = \mathcal{L}(\alpha)$.*

Before we give the actual proof in Section 5.3.3, we discuss some of the consequences of this result. Using Theorem 5.9, it is possible to extend the syntax of $\mathsf{SpLog}_{\mathsf{rx}}$, SpLog, and $\mathsf{EC}^{\mathsf{reg}}$ by defining constraints with vsf-xregex instead of classical regular expressions, without affecting the complexity of evaluation or satisfiability. Naturally, this also allows core spanner representations to use vsf-xregex (e. g. in the definition of relations).

Theorem 5.9 also shows that, given vsf-xregex $\alpha_1, \dots, \alpha_n$, 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 xregex, where even the intersection emptiness problem for two languages is undecidable (cf. Carle and Narendran [5]). An application of this consequence of Theorem 5.9 can be found in Freydenberger and Schmid [18].

5.3.3 Proof of Theorem 5.9

Let α be a vsf-xregex. We first briefly recall a part of the construction that was used in [16] to prove that every language that is generated by a vsf-xregex is also a core spanner language (and, hence, a SpLog-language). There, it is first shown that every vsf-xregex can be expressed as a finite disjunction of xregex paths, where an *xregex path* is a vsf-xregex that is also variable-disjunction free. In other words, an xregex path is a vsf-xregex α such that for each subexpression $(\alpha_1 \vee \alpha_2)$ of α, neither α_1 nor α_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 α_1 and α_2, yielding two vsf-xregex. This process is repeated until each resulting vsf-xregex is also an xregex path. For example, $(x\{\mathtt{a}\} \vee x\{\mathtt{b}\})(y\{\mathtt{c}\} \vee y\{\mathtt{d}\})$ is converted into the four xregex 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 variable-disjunctions*.

Naturally, this can result in an exponential number of xregex paths. As we shall see, SpLog can be used to simulate all these xregex 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-xregex $(x\{\mathtt{a}\} \vee y\{\mathtt{b}\}) \cdot (x\{\mathtt{c}\} \vee y\{\mathtt{d}\}) \cdot \&x \cdot \&y$. There, it is possible to bind each variable once, or one twice and the other not at all, resulting in the words $\mathtt{acc}$, $\mathtt{adad}$, $\mathtt{bccb}$, and $\mathtt{bdd}$ (recall that unbound variables default to ε).

To overcome this, we shall represent each variable x in α 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, $n(x) = n(y) = 2$). For vsf-xregex, this is always bounded by the total number of bindings for x in α. To handle these different variables x_i, we construct a directed acyclic graph $G(\alpha)$ from α that allows us to see how often the value of each variable x can be assigned, and which x_i is accessed by an occurrence of a variable reference $\&x$ (further down, we discuss this idea in more details).

We represent α as a tree $T(\alpha)$, where each node v has a label $\lambda(v)$. If v is a leave, $\lambda(v)$ is a regular expressions or a variable references. If v is an inner node, $\lambda(v)$ is $\vee$, $\circ$, or $x\{\}$ for some $x \in \Xi$. More specifically, if α is a regular expression or an $x \in \Xi$, $T(\alpha)$ consists of one node with label α. 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 $T(\beta)$.

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 $\mathsf{lab}(v)$ is a regular expression or a variable reference, let $G(v) := (V, E)$ with $V := \{v\}$ and $E := \emptyset$, and define $\mathsf{snk}(v) := v$.
– If $\mathsf{lab}(v) = x\{\}$, 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 $\mathsf{lab}(v) \in \{\circ, \vee\}$, 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 an unlabeled new node, and define $\mathsf{snk}(v) = \hat{v}$ and $V := \{v, \hat{v}\} \cup V_l \cup V_R$. Furthermore:
  – If $\mathsf{lab}(v) = \circ$, $E := E_l \cup E_r \cup \{(v, u_l), (\mathsf{snk}(u_l), u_r), (\mathsf{snk}(u_r), \hat{v})\}$.
  – If $\mathsf{lab}(v) = \vee$, $E := E_l \cup E_r \cup \{(v, u), (\mathsf{snk}(u), \hat{v}) \mid u \in \{u_l, u_r\}\}$.

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

Now each path in $G(\alpha)$ from rt to $\mathsf{snk}(\mathsf{rt})$ corresponds to an xregex path that can be derived from α when expanding the variable-disjunctions. This is due to the following reasoning: The process of expanding can be understood as processing $T(\alpha)$ top down. If one encounters a disjunction that contains variable bindings or references, one chooses a side of the disjunction, and discards the other. The obtained xregex path corresponds exactly to the path through $G(\alpha)$ that passes from the nodes of the chosen sides to their 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 rt to v (not including the label of v). Intuitively, $\mathsf{mb}(x, v)$ determines the maximal number of times that a new value can be assigned to x along the path to v.

Moreover, if $\mathsf{lab}(v) = x\{\}$, 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 know that there is a path in $G(\alpha)$, and hence a corresponding xregex path, where this is exactly the i-th binding of x. Recall that by definition, for each subexpression $x\{\beta\}$, we have that β contains neither $x\{\}$ nor $\&x$. For an example, see Figure 5.

Furthermore, for each node, 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 1, and all others have weigth 0), which can be solved in time $O(|V| + |E|)$, cf. Sedgewick and Wayne [42].

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 φ_v. Each of these φ_v has a characteristic free variable y_v that represents the part of W that is created by the sub-xregex that is represented by v. We now define the formulas:

- If $\mathsf{lab}(v)$ is a regular expression, we define $\varphi_v := (y_v \sqsubseteq \mathsf{W}) \wedge \mathsf{C}_{\mathsf{lab}(v)}(y_v)$. This expresses that y_v has to be mapped to a word in $\mathcal{L}(\mathsf{lab}(v))$, and needs no further explanation.
- If $\mathsf{lab}(v) = \&x$, 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 $\mathsf{lab}(v) = x\{\}$, 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 $\mathsf{lab}(v) = \circ$, let u_l and u_r to denote the left and the right child of v in $T(\alpha)$. We define

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

  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 $\mathsf{lab}(v) = \vee$, use u_1 and u_2 to denote the children of v in $T(\alpha)$, without any particular regard to which is left or right. We define

$$X := \{x_i \mid x \in \mathsf{var}(\alpha), 0 \leq i \leq \mathsf{mb}(x, \mathsf{snk}(\mathsf{rt}))\},$$

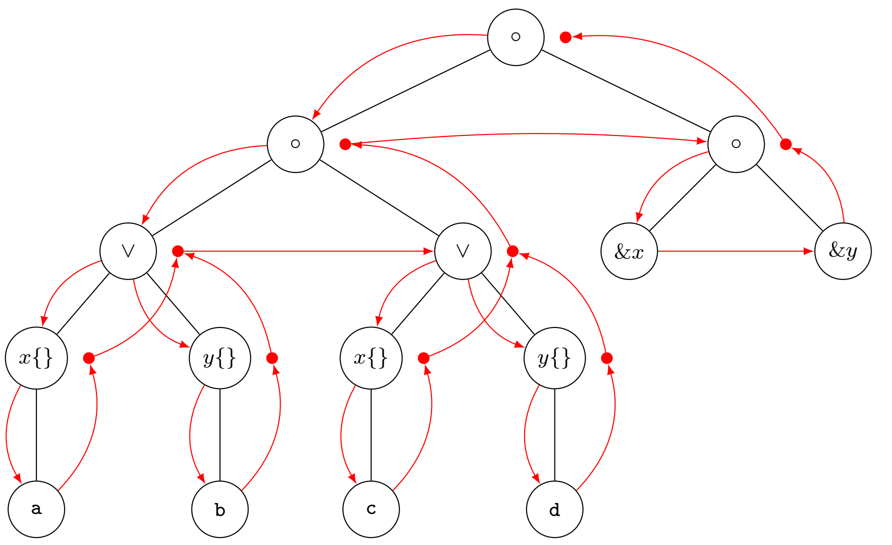


Fig. 4 In black: The tree $T(\alpha)$ for the example $\alpha := ((x\{\mathtt{a}\} \vee y\{\mathtt{b}\}) \cdot (x\{\mathtt{c}\} \vee y\{\mathtt{d}\})) \cdot (\&x \cdot \&y)$ from the proof of Theorem 5.9. In red: The edges and the snk-nodes of $G(\alpha)$. Recall that each node of $T(\alpha)$ is also a node of $G(\alpha)$. There are four different paths from the root to its sink. Each of these paths corresponds to one of the xregex 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 variable-disjunctions in α.

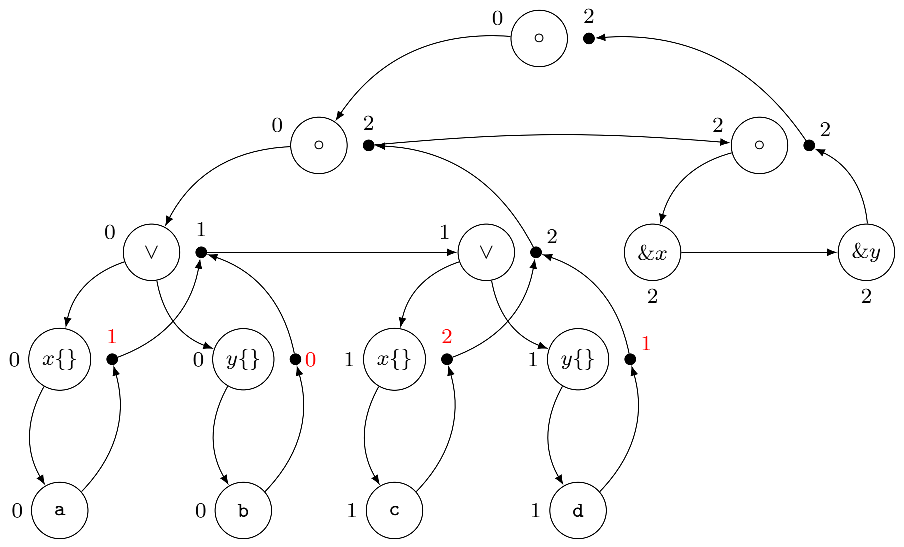


Fig. 5 The graph $G(\alpha)$ for the example from Figure 4. The numbers indicate the value $\mathsf{mb}(x, v)$, where v is the respective node. In order to use this example for the values for $\mathsf{mb}(y, v)$, the numbers that are marked in red have to be changed: From left to right, 1, 0, 2, 1 are replaced with 0, 1, 1, 2, respectively. For all other nodes, $\mathsf{mb}(x, v) = \mathsf{mb}(y, v)$ holds.

and also $m_l^x := \mathsf{mb}(x, \mathsf{snk}(u_l))$ for $l \in \{1,2\}$, and use this for the following formula:

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

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} with $l \in \{1,2\}$, and delegates the task of determining y_{u_l} to φ_{u_l}. Next, the conjunction $\bigwedge_{x_i \in X}(x_i \sqsubseteq W)$ ensures that the formula is safe. 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 φ_{u_1} defines $x_{i+1} = x_i$, $x_{i+2} = x_i$ up to $x_j = x_i$.

The last step of the construction is extending φ_{rt} to the formula

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

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

For both directions of this claim, we observe the following invariant: If v is a node of $T(\alpha)$ with $\mathsf{lab}(v) = \vee$, and $i := \mathsf{mb}(x, v)$ and $j := \mathsf{mb}(x, \mathsf{snk}(v))$, then φ_v assigns exactly the variables x_l with $i < l \leq j$. Each of these assignments can happen either through an equation $x_l = y_u$ (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 an xregex path $\hat{\alpha}$ that is obtained from α by expanding the variable-disjunctions, and $w \in \mathcal{L}(\hat{\alpha})$. As mentioned above, $\hat{\alpha}$ corresponds to a path in $G(\alpha)$ from rt to $\mathsf{snk}(\mathsf{rt})$, which is equivalent to a choice of disjunctions in φ. 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 φ, 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. That side corresponds to a node in $T(\alpha)$, and we can use this to construct the respective path in $G(\alpha)$. This concludes the proof of Theorem 5.9.

6 Limitations of SpLog

While Section 5 discusses various aspects of expressing languages and relations in SpLog, the present section focuses on what SpLog cannot express. Its main part is Section 6.1, where we adapt an inexpressibility result for EC to SpLog. In addition to this, Section 6.2 discusses separating $[\![\mathsf{SpLog}]\!]$ and $[\![\mathsf{EC}^{\mathsf{reg}}]\!]$.

6.1 From EC-Inexpressibility to Non-Selectability for SpLog

In Section 5.1, we defined the notion of SpLog-selectable relations, and examined various relations that are selectable. Our next topic is the opposite: Showing that a relation cannot be selected with SpLog. For this, we shall frequently use the SpLog-inexpressibility of appropriate languages (we defined the notion of SpLog-languages in Section 5.2 – recall in particular that these are defined via the main variable of the formulas). Hence, 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 [16], 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, this left a need for further inexpressibility techniques. 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, Mignosi, and Plandowski [30] developed multiple inexpressibility techniques for EC. Sadly, EC-inexpressibility does not imply SpLog-inexpressibility: For example, if $\Sigma \supseteq \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, one can use the techniques from [30] to show that even the regular language $\{\mathtt{a}, \mathtt{b}\}^*$ is not EC-expressible, although it is obviously SpLog-expressible (like every regular language). On the other hand, while $\mathsf{EC}^{\mathsf{reg}}$-inexpressibility results would be useful, to the author's knowledge, the only result that can be used for this is from Ciobanu, Diekert and Elder [7], namely that every $\mathsf{EC}^{\mathsf{reg}}$-language is an EDT0L-language. In principle, this allows us to use EDT0L-inexpressibility results (of which there are only few; e. g. Ehrenfeucht and Rozenberg [12]), but the comparatively large expressive power of EDT0L limits the usefulness of this approach[7].

But as we shall see, developing a sufficient criterion for EC-expressible SpLog-languages allows us to use one of the techniques from [30] 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 [23]) with observations on EC from [30] yields the following.

Lemma 6.1 *Every bounded regular language is an EC-language.*

[7] A short language theoretic digression that provides a little more context: Every EDT0L-language is an ET0L-language, hence an indexed language (cf. Kari, Rozenberg, and Salomaa [33]), and thereby a context-sensitive language (cf. Mateescu and Salomaa [36]). Although these larger classes haven been studied more intensively than EDT0L, their even larger expressive power makes their inexpressiblity results even less useful for our purposes.

Proof. We base our proof on Theorem 1.1 from [23], 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 class of EC-languages is closed under finite union by definition, every finite language is an EC-language. Closure under concatenation is also straightforward. Finally, as shown in Theorem 5 in [30], for every $w \in \Sigma^*$, w^* is an EC-language. Hence, every bounded regular languages is an EC-language. □

Theorem 6.2 *Every bounded* SpLog*-language is an* EC*-language.*

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 each constraint $\mathsf{C}_A(x)$ in φ can be replaced with bounded regular language $L_{\sqsubseteq}$. Then, Lemma 6.1 states there exists an EC-formula φ_A with $\mathcal{L}(\varphi_A) = L_{\sqsubseteq}$; which means that we can replace each $\mathsf{C}_A(x)$ with $\varphi_A(x)$ without changing the language (these replacements are non-constructive, as we only state the existence of B).

To this end, consider any constraint $\mathsf{C}_A(x)$ in φ, together with a substitution σ that is obtained from a substitution $\hat{\sigma} \models \varphi$. As φ 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, we have $\sigma(\mathsf{W}) \in B$, as $\mathcal{L}(\varphi) \subseteq B$.

As φ is a $\mathsf{SpLog}(\mathsf{W})$-formula, $\sigma(x) \sqsubseteq \sigma(\mathsf{W})$, which implies $\sigma(x) \in B_{\sqsubseteq}$, where $B_{\sqsubseteq} := \{u \mid u \sqsubseteq v \text{ for some } v \in B\}$. Hence, $\sigma(x) \in L_{\sqsubseteq}$, where we define $L_{\sqsubseteq} := \mathcal{L}(A_{\sqsubseteq}) \cap B_{\sqsubseteq}$. Less formally, we observe that the constraint C_A does not actually use all of $\mathcal{L}(A)$, but just the words from $L_{\sqsubseteq}$. All that remains to be shown is that this language is bounded regular, as then Lemma 6.1 applies.

Observe that B is a regular language; and as the class of regular languages is closed under taking the set of all subwords (a common exercise), this means that $B_{\sqsubseteq}$ is regular as well. The class of regular languages is also closed under intersection; thus, $L_{\sqsubseteq}$ is regular. It is also bounded, as every set of subwords of a bounded language is bounded (Lemma 5.1.1 in Ginsburg [22]). □

The intuition behind this is very simple: In SpLog, every variable is a subword of the main variable. Hence, if the formula defines a bounded language, the constraints of the variables also have to “fit into” the bounded language, which means that they can be replaced with a bounded regular language, which is an EC-language (due to Lemma 6.1). This reasoning does not generalize to $\mathsf{EC}^{\mathsf{reg}}$, as that logic does not restrict variables to subwords (hence, the variables do not inherit the boundedness of the language).

The EC-inexpressibility technique from [30] that we are going to use is based on the following definition by Karhumäki, Plandowski, and Rytter [31].

Definition 6.3 A word $w \in \Sigma^+$ is *imprimitive* if there exist a word $u \in \Sigma^+$ and $n \geq 2$ with $w = u^n$. Otherwise, w is *primitive*. For a 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 $0 < i < k$.

Finally, we define $\exp_Q(w) := \max(T_Q(w) \cup \{0\})$, where

$$T_Q(w) := \{x \mid Q^x \text{ occurs in the } \mathscr{F}_Q\text{-factorization of } w\}.$$

For every primitive word Q, the $\mathscr{F}_Q$-factorization of every word w and $\exp_Q(w)$ are uniquely defined (cf. [30,31]). We use this definition in the following pumping result for EC.

Theorem 6.4 (Karhumäki et al. [30]) *For every EC-language L and every primitive word Q, there exists $k \geq 0$ such that, for each $w \in L$ with $\exp_Q(w) > k$, there is a word $u \in L$ with $\exp_Q(u) \leq k$ which is obtained from w by removing some occurrences of Q.*

Combining this with Theorem 6.2, we immediately obtain the following pumping result for SpLog (and, hence, core spanners).

Theorem 6.5 *For every bounded SpLog-language L and every primitive word Q, there exists $k \geq 0$ such that, for each $w \in L$ with $\exp_Q(w) > k$, there is a word $u \in L$ with $\exp_Q(u) \leq k$ which is obtained from w by removing some occurrences of Q.*

Example 6.6 As shown by Fagin et al. [13] (Theorem 4.21), $L_{\mathsf{el}} := \{\mathtt{a}^i\mathtt{b}^i \mid i \geq 0\}$ is not expressible with core spanners. The length of this proof is roughly one page in the style of Journal of the ACM.

Contrast this to the following: Assume that L_{el} is a SpLog-language. Choose the primitive word $Q := \mathtt{a}$. Then there exists $k \geq 0$ that satisfies Theorem 6.5. Choose $w := \mathtt{a}^{k+2}\mathtt{b}^{k+2}$, and observe that $\exp_Q(w) = k+1 > k$, which is due to the factorization $w = \varepsilon \cdot \mathtt{a}^{k+1} \cdot \mathtt{a}\mathtt{b}^{k+2}$. Hence there exists a word $u = \mathtt{a}^{k+2-j}\mathtt{b}^{k+2}$, $j > 0$, with $u \in L_{\mathsf{el}}$. 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 $R_{\mathsf{el}} = \{(u,v) \mid |u| = |v|\}$ is not selectable with core spanners. Expressed with SpLog instead of spanners, the argument is that otherwise, $\mathcal{L}(\varphi) = L_{\mathsf{el}}$ for $\varphi(\mathsf{W}) := \exists x, y \colon (\mathsf{W} = xy \wedge \mathsf{C}_{\mathtt{a}^*}(x) \wedge \mathsf{C}_{\mathtt{b}^*}(y) \wedge R_{\mathsf{el}}(x,y))$.

Note that Karhumäki et al. [30] and Ilie [28] use the same approach (show the non-selectability of a relation by proving that a suitable language is not expressible) to show that R_{el} and various other relations are not selectable with EC (in particular, they also use L_{el} and Theorem 6.4 for R_{el}). Before we use this technique to prove that some other relations are not SpLog-selectable, we introduce a few more definitions: For every word $w \in \Sigma^*$, its *reversal* w^R is the word that is obtained by reading w from right to left. For $x, y \in \Sigma^*$, we say that x is a *scattered subword* of y if there exist $k \geq 1$ and words $x_1, \ldots, x_k, y_0, \ldots y_k \in \Sigma^*$ such that $x = x_1 \cdots x_k$ and $y = y_0(x_1y_1)\cdots(x_ky_k)$.

Proposition 6.7 *Consider the following binary relations over Σ^*:*

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

None of these relations is SpLog-selectable.

Proof. The proof follows the same outline as Example 6.6: We first define three languages L_1 to L_3, each of which is shown not to be a SpLog-language. For each relation, we then show that SpLog-selectability of this relation implies $L_i \in \mathcal{L}(\mathsf{SpLog})$ for some i. We choose distinct $\mathtt{a}, \mathtt{b}, \in \Sigma$, and define

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

Each of these three languages is bounded. Hence, we can use Theorem 6.5 to show that they are not SpLog-languages.

ad L_1: This proof is almost identical to the example: Assume that L_1 is a SpLog-language, and choose $Q_1 := \mathtt{b}$. Then there exists some k_1 that satisfies Theorem 6.5. Let $w_1 := \mathtt{a}^{k_1+2}\mathtt{b}^{k_1+2}$, and observe the $\mathscr{F}_{Q_1}$-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, $L_1 \notin \mathcal{L}(\mathsf{SpLog})$.

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 $\mathscr{F}_{Q_2}$-factorization $w_2 = \mathtt{a}^{k+2}\mathtt{ba} \cdot (\mathtt{ba})^{k+1} \cdot \varepsilon$.

ad L_3: Assume that L_3 is a SpLog-language, and choose $Q_3 := \mathtt{abaabb}$. Let k_3 be the constant from Theorem 6.5, and choose $w_3 := (\mathtt{abaabb})^{k_3+2}(\mathtt{bbaaba})^{k_3+2}$. The $\mathscr{F}_{Q_3}$-factorization is $w_3 = \varepsilon \cdot (\mathtt{abaabb})^{k_3+1} \cdot \mathtt{abaabb}(\mathtt{bbaaba})^{k_3+2}$; hence, $\exp_{Q_3}(w_3) = k_3 + 1$. By Theorem 6.5, there is some $j > 0$ such that $u_3 \in L_3$ for $u_3 := (\mathtt{abaabb})^{k_3+2-j}(\mathtt{bbaaba})^{k_3+2}$. Contradiction. Thus, $L_3 \notin \mathcal{L}(\mathsf{SpLog})$.

Using the languages: Assume some relation R out of R_{scatt}, $R_{\mathsf{num}(a)}$, R_{permut}, R_{rev}, and $R_{<}$ is selected by $\varphi_R(\mathsf{W}; x, y) \in \mathsf{SpLog}$. We then define:

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

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. □

To our inconvenience, the restriction to bounded languages limits the applicability of this approach. For example, Ilie [28] shows that over a two letter alphabet, the language of square-free words (i. e., words that contain no subword xx with $x \neq \varepsilon$) is not an EC-language. Although one might conjecture that it is also not a SpLog-language, one can easily see that every bounded subset of this language has to be finite, which means that our technique fails.

Furthermore, consider the relation $R_{\mathsf{pow}} := \{(x, x^n) \mid x \in \Sigma^*, n \geq 1\}$. It was already conjectured in [16] that R_{pow} is not SpLog-selectable; but there is no suitable bounded language that could be used to prove this.

Another example where this approach fails is the *uniform-0-chunk language* $L_{\mathsf{uzc}} := \mathcal{L}(\alpha_{\mathsf{uzc}})$, which is defined through the xregex (see Section 5.3) $\alpha_{\mathsf{uzc}} := 1^+ x\{0^*\}(1^+ \& x)^* 1^+$. Intuitively, in every word of L_{uzc}, all 0-chunks (maximal subwords from 0^*) have the same length. This language was used in [13] to prove that the relation $\not\sqsubseteq$ is not selectable with core spanners. Clearly, L_{uzc} is not bounded, and intersecting it with a bounded languages limits us to a bounded number of 0-chunks in every word, or to 0-chunks of a bounded length (thus obtaining a regular language). Hence, this approach fails for L_{uzc}.

6.2 Comparing the Power of SpLog and $\mathsf{EC}^{\mathsf{reg}}$

A question that remains open in this paper is whether $[\![\mathsf{EC}^{\mathsf{reg}}]\!] = [\![\mathsf{SpLog}]\!]$. We briefly address some aspects of an open subproblem, namely whether $\mathcal{L}(\mathsf{EC}^{\mathsf{reg}}) = \mathcal{L}(\mathsf{SpLog})$. While one might conjecture that $\mathsf{EC}^{\mathsf{reg}}$ is more powerful, our proof that the class of SpLog-languages is closed under right quotient $/a$ (Lemma 5.7) serves as an example of where SpLog replicates behavior of $\mathsf{EC}^{\mathsf{reg}}$; although with significant extra effort.

The right quotient $/a$ can be seen as a variant of the prefix operator, but closure under the latter is more complicated than for the former. In fact, the question whether $\mathcal{L}(\mathsf{SpLog})$ is closed under the prefix operator is inherently related to the question whether SpLog and $\mathsf{EC}^{\mathsf{reg}}$ can define the same languages.

Proposition 6.8 $\mathcal{L}(\mathsf{EC}^{\mathsf{reg}}) = \mathcal{L}(\mathsf{SpLog})$ *if and only if* $\mathcal{L}(\mathsf{SpLog})$ *is closed under the prefix operator.*

Proof. 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 $\psi(x) := \exists y, z \colon ((y = xz) \wedge \varphi(y))$. 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 $\mathsf{EC}^{\mathsf{reg}}$-language. As we assumed $\mathcal{L}(\mathsf{EC}^{\mathsf{reg}}) = \mathcal{L}(\mathsf{SpLog})$, it is also a SpLog-language.

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 [10] (also see the remark at the end of Section 2.1), φ can be converted into an equivalent $\mathsf{EC}^{\mathsf{reg}}$-formula $\chi = \exists \vec{y}\colon (\eta \wedge C)$ where $\vec{y}$ is a sequence of variables, and C is a conjunction of constraints.

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

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

Now, L_1 is the result of applying the prefix operator to the SpLog-language $\mathcal{L}(\psi)$; which means that L_1 is a SpLog-language due to our initial assumption. As SpLog-languages are closed under intersection with regular languages (by simply adding the corresponding regular constraint), L_2 is a SpLog-language; and so is L_3 (due to Lemma 5.7). We conclude $L_3 = \{\sigma(x) \mid \sigma \models \chi\} = \mathcal{L}(\chi) = \mathcal{L}(\varphi)$. Hence, $\mathcal{L}(\varphi) \in \mathcal{L}(\mathsf{SpLog})$. □

To avoid potential confusion, recall that although we showed in Example 5.3 that the prefix relation is SpLog-selectable, this does not mean that we can use this to turn a SpLog-formula for some language L into a formula for the language of all prefixes of L.

In principle, Proposition 6.8 could offer an elegant way of (dis-)proving $\mathcal{L}(\mathsf{SpLog}) = \mathcal{L}(\mathsf{EC}^{\mathsf{reg}})$ by (dis-)proving that the former class is closed under the prefix operator. In practice, this seems to be more of indicator that (dis-)proving closure under the prefix operator is hard.

The question whether $\mathcal{L}(\mathsf{SpLog}) = \mathcal{L}(\mathsf{EC}^{\mathsf{reg}})$ seems to be surprisingly complicated; even when only considering only word equations without constraints: 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 [9], Karhumäki and Saarela [32]). In all cases that the author considered, it turned out that one could 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 [9], Ilie and Plandowski [29]). On the other hand, these constructions do not appear to generalize straightforwardly to an equivalence proof.

We conclude this section with a consequence that $\mathcal{L}(\mathsf{EC}^{\mathsf{reg}}) \neq \mathcal{L}(\mathsf{SpLog})$ would have. To prove this, we combine Lemma 5.7 with the following result that is commonly known as Greibach's Theorem (originally from Greibach [24], this formulation is Theorem 8.14 in Hopcroft and Ullman [27]).

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 Σ. Let P be any non-trivial property that is true for all regular languages and that is preserved under $/a$, where $a \in \Sigma$. Then P is undecidable for $\mathcal{C}$.*

Proposition 6.9 *Assume $\mathcal{L}(\mathsf{EC}^{\mathsf{reg}}) \neq \mathcal{L}(\mathsf{SpLog})$. Given $\varphi \in \mathsf{EC}^{\mathsf{reg}}$, it is undecidable whether $\mathcal{L}(\varphi) \in \mathcal{L}(\mathsf{SpLog})$.*

Proof. Assume that $\mathcal{L}(\mathsf{EC}^{\mathsf{reg}}) \neq \mathcal{L}(\mathsf{SpLog})$. To use Greibach's Theorem, we choose the class of $\mathsf{EC}^{\mathsf{reg}}$-languages for $\mathcal{C}$, the property "L is a SpLog-language" for P. We discuss the conditions of Greibach's Theorem step by step: The class of SpLog-languages is effectively closed under concatenation and union: Given $\varphi_1, \varphi_2 \in \mathsf{EC}^{\mathsf{reg}}$, we have $\mathcal{L}(\varphi_1 \vee \varphi_2) = \mathcal{L}(\varphi_1) \cup \mathcal{L}(\varphi_2)$ and $\mathcal{L}(\varphi_c) = \mathcal{L}(\varphi_1) \cdot \mathcal{L}(\varphi_2)$ for $\varphi_c := \exists u, v \colon (mv = u \cdot v) \wedge \varphi_1(u) \wedge \varphi_2(v)$. Recall that $\mathsf{EC}^{\mathsf{reg}}$ includes all regular languages, which gives us effective closure under concatenation with regular languages.

If $|\Sigma| \geq 2$, then $\mathcal{L}(\varphi) = \Sigma^*$ is undecidable when given $\varphi \in \mathsf{EC}^{\mathsf{reg}}$ as input: This follows (even for $\varphi \in \mathsf{SpLog}$) for example from Theorem 5.9 and the undecidability of this problem for vsf-xregex (see [14]). An alternative proof is discussed in Section 7.4.

Next, P is a non-trivial property: The class of SpLog-languages is not empty, and $\mathcal{L}(\mathsf{SpLog}) \neq \mathcal{L}(\mathsf{SpLog})$ holds by our assumption. Every regular language is also a SpLog-language, and $\mathcal{L}(\mathsf{SpLog})$ is closed under $/a$ according to Lemma 5.7. Hence, if $\mathcal{L}(\mathsf{EC}^{\mathsf{reg}}) \neq \mathcal{L}(\mathsf{SpLog})$, Greibach's Theorem applies; which means that $\mathcal{L}(\varphi) \in \mathcal{L}(\mathsf{SpLog})$ is undecidable for $\varphi \in \mathsf{SpLog}$. □

7 Conjunctive Path Queries on Marked Paths

In this section, we examine the connection between SpLog and a querying language for graphs, namely unions of conjunctive regular path queries (UCRPQs) that are extended with string equalities. In the conference version [15] of this paper, this section was a short paragraph that mostly consisted of the following claim: "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." (Recall that we defined prenex conjunctions and DPC-normal form in Section 5.2.)

But this claim was overly optimistic. In fact, if taken literally, it is wrong; although we shall see that it holds for a rich and natural class of restricted queries. But explaining this adequately requires further definitions, and the author apologizes to the reader for burdening them with even more notation.

This section is structured as follows: First, we introduce UCRPQs in Section 7.1. We then discuss the notion of marked paths, and how graph queries on marked paths connect to SpLog in Section 7.2. Finally, Section 7.3 states the transformations between these queries and SpLog, and Section 7.4 briefly discusses how this can be used to extend previous undecidability results.

7.1 Conjunctive Regular Path Queries with Equality

We begin with the definition of the data model. Let Δ be a terminal alphabet. A *Δ-labeled db-graph* is a directed graph $G = (V, E)$, where V is a finite set of nodes, and $E \subseteq V \times \Delta \times V$ is a finite set of edges with labels from Δ. A *path* p between two nodes $v_0, v_n \in G$ with $n \geq 1$ is a sequence

$$p = (v_0, a_1, v_1)(v_1, a_2, v_2) \cdots (v_{n-1}, a_n, v_n)$$

of edges $(v_{i-1}, a_i, v_i) \in E$, and we define its *label* $\mathsf{lab}(p) := a_1 a_2 \cdots a_n$ as the word that is formed by the labels along the edges of p. We also define the *empty path* (v, ε, v) for every $v \in V$, with $\mathsf{lab}(v, \varepsilon, v) = \varepsilon$.

A *regular path query (RPQ)* is a query of the form $\varphi(x, y) = (x, L, y)$, where the variables x and y range over nodes, and L is a regular language; and $[\![\varphi]\!](G)$ contains exactly those pairs of nodes (x, y) for which there is a path p from x to y in G such that $\mathsf{lab}(p) \in L$. By considering conjunctions of RPQs, one obtains *conjunctive regular path queries (CRPQs).* Barceló, Libkin, Lin, and Wood [2] introduced *extended regular path queries (ECRPQs)*, which extend CRPQs by allowing comparisons of path labels via regular relations, like string equality and the equal lengths relation. In this paper, we follow Fagin et al. [13], by considering a class of queries between CRPQ and ECRPQ, namely conjunctive regular queries with string equality predicates.

The following definition of these queries is based on the definition of ECRPQs by Barceló et al. [2].

Definition 7.1 A *conjunctive regular path query with string equalities (equality CRPQ)* over the alphabet Δ is a formula

$$\varphi(\vec{z}_f) = \exists \vec{z}_b \colon \bigwedge_{1 \leq i \leq m} (x_i, \pi_i \colon L_i, y_i) \wedge \bigwedge_{1 \leq j \leq n} (\xi_j^L = \xi_j^R)$$

such that $m \geq 1$, $n \geq 0$, and

1. all $x_1, \ldots, x_m$ and $y_1, \ldots, y_m$ are node variables (and not necessarily distinct); the set of these variables is denoted by $\mathsf{NVars}(\varphi)$,
2. $\pi_1, \ldots, \pi_m$ are pairwise distinct path variables, the set of these is denoted by $\mathsf{PVars}(\varphi)$,
3. the L_i are regular languages over Δ that are defined by NFAs or regular expressions, and we call L_i the *range* of π_i,
4. the ξ_j^L and ξ_j^R are path variables from $\mathsf{PVars}(\varphi)$,
5. $\vec{z}_f$ is a tuple of variables from $\mathsf{NVars}(\varphi)$; these are the *free variables* of φ, and their set is denoted by $\mathsf{free}(\varphi)$,
6. $\vec{z}_b$ is a tuple that contains exactly the variables of $\mathsf{NVars}(\varphi) - \mathsf{free}(\varphi)$.

We use $\mathsf{CRPQ}^=$ to denote the class of all equality CRPQs, and $\mathsf{CRPQ}^=_{\mathsf{rx}}$ to denote the subclass of that defines all L_j only by using regular expressions.

For every Δ-labeled db-graph $G = (V, E)$ and every mapping $\tau \colon \mathsf{free}(\varphi) \to V$, we define that $(\tau, G) \models \varphi$ if there exist a mapping τ' from $\mathsf{NVars}(\varphi)$ to V and a mapping μ from $\mathsf{PVars}(\varphi)$ to paths in G such that:

1. $\tau'(x) = \tau(x)$ for all $x \in \mathsf{free}(\varphi)$,
2. $\mu(\pi_i)$ is a path from $\tau'(x_i)$ to $\tau'(y_i)$ for all $1 \leq i \leq m$,
3. $\mathsf{lab}(\mu(\pi_i)) \in L_i$ for all $1 \leq i \leq m$,
4. $\mathsf{lab}(\mu(\xi_j^L)) = \mathsf{lab}(\mu(\xi_j^R))$ for all $1 \leq j \leq n$.

Based on this, we define $[\![\varphi]\!](G)$ as the set of all τ with $(\tau, G) \models \varphi$.

Intuitively, all variables are quantified existentially, and the words formed by the labels along the paths have to belong to the respective languages or satisfy the respective string equalities.

An important difference between Definition 7.1 and the definition of ECR-PQs from Barceló et al. [2] is that we assume that all path variables are bound. We shall only consider path queries on very restricted graphs, where all paths are uniquely identified by their first and last node. This allows us to streamline the definition. We also use the shorthand notation (x, L, y) instead of $(x, \pi\colon L, y)$, if π is not used in any equality check.

Example 7.2 Consider the following equality CRPQ:

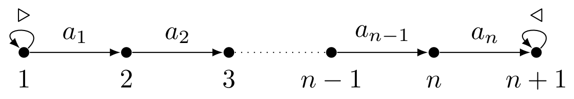
$$\varphi(x,y) := \exists z_1, z_2\colon$$
$$(x, \pi_1\colon (\mathsf{aa})^+, z_1) \wedge (z_1, \mathsf{b}, z_2) \wedge (z_2, \pi_3\colon (\mathsf{aaa})^+, y) \wedge (\pi_1 = \pi_3).$$

Then for every db-grap G, we have that $[\![\varphi]\!](G)$ contains exactly those pairs of nodes (x, y) of G for which there exists a path π from x to y such that $\mathsf{lab}(\pi)$ is from the language $\mathsf{a}^{6i}\mathsf{b}\mathsf{a}^{6i}$, $i \geq 1$. Nodes and edges may occur multiple times along π. ◇

Another model that was examined by Fagin et al. [13] are *unions of equality CRPQs (short: equality UCRPQs)*. An equality UCRPQ is a formula $\varphi = \bigvee_{i=1}^{k} \varphi_i$, where $\varphi_i \in \mathsf{CRPQ}^=$ for all $1 \leq i \leq k$, and all φ_i have the same free variables. Consequently, we define $[\![\varphi]\!](G) := \bigcup_{i=1}^{k} [\![\varphi_i]\!](G)$; and we use $\mathsf{UCRPQ}^=$ to denote the class of all equality UCRPQs, and $\mathsf{UCRPQ}^=_{\mathsf{rx}}$ for the subclass that defines ranges only with regular expressions.

7.2 Marked Paths

Obviously, any attempt to compare SpLog (or spanners) with path queries must overcome the basic problem that the former query strings, while the latter query graphs. As a solution, Fagin et al. [13] proposed that the input of the path queries is restricted to *marked paths*. The marked path for a word $w = a_1 \cdots a_n$ with $n \geq 0$ is the db-graph $\mathsf{G}^w_{\mathsf{mp}}$ over the extended alphabet $\Delta := \Sigma \cup \{\triangleright, \triangleleft\}$ that consists of the nodes 1 to $n+1$, and an edges with label a_i from i to $i+1$ for each $1 \leq i \leq n$. Furthermore, there is a loop with the special symbol $\triangleright$ on the node 1, and a loop with the special symbol $\triangleleft$ on the node $n+1$. This is depicted in the following illustration:



Fagin et al. [13] point out that the markings can be used to identify the first and last node of the marked path using the RPQs $(x, \triangleright, x)$ and $(x, \triangleleft, x)$, respectively.

As shown in [13], every core spanner on input w can be expressed by an equality UCRPQ on the marked path $\mathsf{G}^w_{\mathsf{mp}}$, by using two node variables $x^{\vdash}$ and $x^{\dashv}$ for every span variable x. These variables represent the start and the end of the span, and every node assignment τ translates into the span $[\tau(x^{\vdash}), \tau(x^{\dashv})\rangle$.

Likewise, [13] showed that every equality UCRPQ that expresses a span in this way can also be transformed into an equivalent core spanner representation. The transformations were not considered with respect to their complexity, but as we shall prove, some are impossible in polynomial time (unless $\mathsf{P} = \mathsf{NP}$).

But first, note that using SpLog as a framework allows us to define a more convenient notion of "simulating a path query", as we can represent each node i on a marked path $\mathsf{G}^w_{\mathsf{mp}}$ in SpLog as the prefix of w that has length i. This is used in the following definition.

Definition 7.3 Let $\varphi \in \mathsf{UCRPQ}^{=}$ and $\psi \in \mathsf{SpLog}(\mathsf{W})$ with $\mathsf{free}(\varphi) = \mathsf{free}(\psi) - \{\mathsf{W}\}$. We say that ψ *realizes* φ *(on marked paths)* if for all $w \in \Sigma^*$, we have $\sigma \in [\![\psi]\!](w)$ if and only if $\tau \in [\![\varphi]\!](\mathsf{G}^w_{\mathsf{mp}})$ with $w_{[1,\tau(x)\rangle} = \sigma(x)$ for all $x \in \mathsf{free}(\varphi)$.

Building on this definition, we can compare arbitrary equality UCRPQs on marked paths to SpLog, instead of being restricted to those that simulate spanners (this notion also extends to any type of query that maps db-graphs to sets of node assignments, but this is outside the scope of the present paper).

For the other direction, we combine Definition 4.7 with the encoding of spanners in path queries from [13] that was mentioned above.

Definition 7.4 Let $\varphi \in \mathsf{SpLog}(\mathsf{W})$ and $\psi \in \mathsf{UCRPQ}^{=}$ with

$$\mathsf{free}(\psi) = \{x^{\vdash}, x^{\dashv} \mid x \in (\mathsf{free}(\varphi) - \{\mathsf{W}\})\}.$$

Then ψ *realizes* φ if, for all $w \in \Sigma^*$ and all substitutions σ, we have $\tau \in [\![\psi]\!](\mathsf{G}^w_{\mathsf{mp}})$ if and only if $\sigma \in [\![\varphi]\!](w)$ with $\sigma(x) = w_{[\tau(x^{\vdash}),\tau(x^{\dashv})\rangle}$ for all $x \in \mathsf{free}(\psi)$.

With these definitions, we can directly adapt the notion of polynomial time conversions (recall Section 4.1) to queries on marked paths.

Our next step is proving that equality UCRPQs on marked paths are too powerful to allow polynomial time transformations to SpLog. But as we shall see in the other results of this section, this is arguably due to side effects of the encoding, and not an inherent succinctness advantage of $\mathsf{CRPQ}^{=}$ over SpLog.

More specifically, we can prove that the existence of a polynomial time transformation from $\mathsf{CRPQ}^{=}$ to SpLog implies $\mathsf{P} = \mathsf{NP}$. To show this, we shall abuse the loop with the start marker $\triangleright$ to encode the NP-hard non-emptiness problem for regular expressions over unary alphabets[8].

[8] Finding a citation for this turned out to be surprisingly hard: It is a well-known consequence of Stockmeyer and Meyer [43] that the intersection emptiness problem for an

Lemma 7.5 (Neven and Martens [35]) *Given regular expressions $\alpha_1, \dots \alpha_k$ over the alphabet $\{\mathsf{a}\}$, deciding whether $\emptyset \neq \bigcap_{i=1}^{k} \mathcal{L}(\alpha_i)$ is NP-hard.*

Lemma 7.5 directly allows us to state the following lower bound result on the evaluation of equality CRPQs. Note that it holds for any fixed marked path, even the marked path $\mathsf{G}^{\varepsilon}_{\mathsf{mp}}$.

Lemma 7.6 *Fix $w \in \Sigma^*$. Given $\varphi \in \mathsf{CRPQ}^{=}_{\mathsf{rx}}$, deciding $[\![\varphi]\!](\mathsf{G}^{w}_{\mathsf{mp}}) \neq \emptyset$ is NP-hard.*

Proof. We prove this via reduction from the non-emptiness problem for regular expressions over unary alphabets, see Lemma 7.5. Given regular expressions $\alpha_1, \dots, \alpha_k$ over the unary terminal alphabet $\{\triangleright\}$, we define the equality CRPQ

$$\varphi(x) :=: \bigwedge_{i=1}^{k} (x, \pi_i \colon \alpha_i, x) \wedge \bigwedge_{j=2}^{k} (\pi_1 = \pi_j).$$

Then for every marked path $\mathsf{G}^{w}_{\mathsf{mp}}$, we have $[\![\varphi]\!](\mathsf{G}^{w}_{\mathsf{mp}}) \neq \emptyset$ if and only if there is an $n \geq 0$ with $\triangleright^n \in \bigcap_{i=1}^{k} \mathcal{L}(\alpha_i)$. □

As SpLog-formulas can be evaluated in polynomial time on the input ε (see the proof of Lemma 4.8), Lemma 7.6 immediately leads us to the following.

Proposition 7.7 $\mathsf{P} = \mathsf{NP}$*, if there is a polynomial time conversion from* $\mathsf{CRPQ}^{=}_{\mathsf{rx}}$ *on marked paths to* SpLog.

The proof of Lemma 7.6 shows that the encoding of words in marked paths causes problems for the transformation, as we have to account for arbitrarily long blocks of the marker symbols $\triangleright$ and $\triangleleft$.

One way of dealing with this problem is to prevent the use of equality checks on path variables that can contain these marker symbols (which was proposed by Fagin et al. [13]). In fact, this is the approach that we shall choose; but before we do that in Section 7.3, we briefly discuss an alternative way of encoding words in paths, which we call *straight marked paths.* Instead of using loops on the first and last nodes (like the marked paths do), straight marked paths and an additional initial and final node. Hence, the straight marked path $\mathsf{G}^{w}_{\mathsf{smp}}$ of a word $w = a_1 \cdots a_n$ is this graph:

$$\begin{tikzcd}
0 \arrow[r, "\triangleright"] & 1 \arrow[r, "a_1"] & 2 \arrow[r, "a_2"] & 3 \arrow[r, dotted, no head] & n-1 \arrow[r, "a_{n-1}"] & n \arrow[r, "a_n"] & n+1 \arrow[r, "\triangleleft"] & n+2
\end{tikzcd}$$

In particular, the graph $\mathsf{G}^{\varepsilon}_{\mathsf{smp}}$ that encodes the empty word as a straight marked path consists of the path $(0, \triangleright, 1), (1, \triangleleft, 2)$. In contrast to marked paths, straight marked paths do not allow queries to assign paths of arbitrary length. But as the proof of the next result demonstrates, this encoding causes new problems.

unbounded number of DFAs is PSPACE-complete. It seems to be less well-known that as a consequence of Galil [20], the problem is NP-complete over unary terminal alphabets (but locating the proof in that paper without already knowing the main idea can be rather difficult). Luckily, Lemma 27 in Neven and Martens [35] provides an explicit and accessible proof (and although it covers only the automata case, it directly translates to regular expressions).

Proposition 7.8 *Given $\varphi \in \mathsf{CRPQ}^{=}_{\mathsf{rx}}$, deciding if $[\![\varphi]\!](\mathsf{G}^{\varepsilon}_{\mathsf{smp}}) \neq \emptyset$ is NP-hard.*

Proof. We show this via a reduction from the problem *one-in-three satisfiability*, which is defined as follows: Given a set M and non-empty subsets $S_1, \ldots, S_k \subseteq M$ such that $|S_i| \leq 3$ for all i, is there a subset $T \subseteq M$ with $|S_i \cap T| = 1$ for all i? As shown by Schaefer [40], this problem is NP-complete.

Assume that each S_i consists of $s_{i,1} \ldots, s_{i,|S_i|}$. The main idea is to represent each $s_{i,j}$ with a path variable $\pi_{i,j}$ that is mapped to the path $(0, \triangleright, 1)$ if $s_{i,j} \in T$, and to an empty path on 0 or 1 otherwise. Equality tests are used to ensure that all $s_{i,j}$ and $s_{i',j'}$ with $s_{i,j} = s_{i',j'}$ have consistent assignments. Following this intuition, we define

$$\begin{aligned}\varphi(x_0, x_1, y_1, \ldots, y_k, z_1, \ldots, z_k) &:= (x_0, \triangleright, x_1)\\ &\wedge \bigwedge_{i_1}^{k} \big((x_0, \pi_{i,j} \colon L_{i,1}, y_i) \wedge (y_i, \pi_{i,j} \colon L_{i,2}, z_i) \wedge (z_i, \pi_{i,j} \colon L_{i,2}, x_1)\big)\\ &\qquad\qquad \wedge \bigwedge_{i=1}^{k} \bigwedge_{j=1}^{|S_i|} \bigwedge_{s_{i',j'} = s_{i,j}} (\pi_{i,j} = \pi_{i',j'}),\end{aligned}$$

where $L_{i,j} := \{\varepsilon\}$ if $j > |S_i|$, and $L_{i,j} := \{\varepsilon, \triangleright\}$ if $j \leq |S_i|$. Clearly, φ can be constructed in polynomial time. To see that it is correct, note that on $\mathsf{G}^{\varepsilon}_{\mathsf{smp}}$, the node variables x_0 and x_1 have to map to the nodes 0 and 1, respectively. Then the conjunctions in the middle row of the definition of φ ensure that for each S_i, exactly one $\pi_{i,j}$ is set to the path from 0 to 1. Take particular note that if $j > |S_i|$, then $\pi_{i,j}$ must map to the empty path (on the node 0 or 1).

Hence, there is a one-to-one correspondence between node assignments $\tau \in [\![\varphi]\!](\mathsf{G}^{\varepsilon}_{\mathsf{smp}})$, and sets T that are solutions of the one-in-three satisfiability problem; which means that $[\![\varphi]\!](\mathsf{G}^{\varepsilon}_{\mathsf{smp}}) \neq \emptyset$ if and only if such a set T exists. □

Hence, Proposition 7.7 also holds for $\mathsf{CRPQ}^{=}_{\mathsf{rx}}$ on straight marked paths (if we extend Definition 7.4 appropriately). The author considers this a sign that changing the encoding of the string does not overcome the encoding issues (at least not when using obvious encodings). Instead, the next section follows the example of Fagin et al. [13] and restricts the queries.

7.3 Conversions Between $\mathsf{UCRPQ}^{=}$ and SpLog

We saw in the previous section that the special symbols $\triangleright$ and $\triangleleft$ can be problematic when they occur in the languages of path variables that are compared with equalities. This was already observed by Fagin et al. [13] (although not from a complexity point of view). To overcome technical difficulties in the transformation of path queries to spanners, Fagin et al. proposed the notion of *Σ-restriced equality UCRPQs*, which can only compare paths that do not have the special markers as labels.

Definition 7.9 A path variable in an equality CRPQ φ is *Σ-restricted* if its range is a subset of Σ^* (i. e., no word in the range contains $\triangleright$ or $\triangleleft$). An equality $(\pi = \rho)$ in φ is Σ-restricted if π and ρ are Σ-restricted. Finally, φ is Σ-restricted if all of its equalities are Σ-restricted; and an equality UCRPQ is Σ-restricted if all of its underlying equality CRPQs are Σ-restricted.

Clearly, one can check in polynomial time whether an equality UCRPQ is Σ-restricted. Moreover, as shown in [13], every equality UCRPQ can be converted into a Σ-restricted equality URCPQ that is equivalent on marked paths. But we can conclude from Lemma 7.6 that this transformation is not possible in polynomial time (under the assumption that $\mathsf{P} \neq \mathsf{NP}$).

Lemma 7.10 *Assume that there is an algorithm that, given $\varphi \in \mathsf{CRPQ}^{=}_{\mathsf{rx}}$, computes in polynomial time a Σ-restricted $\psi \in \mathsf{UCRPQ}^{=}$ with $[\![\psi]\!](\mathsf{G}^{w}_{\mathsf{mp}}) = [\![\varphi]\!](\mathsf{G}^{w}_{\mathsf{mp}})$ for all $w \in \Sigma^*$. Then $\mathsf{P} = \mathsf{NP}$.*

Proof. This follows directly from Lemma 7.6, and the fact that for Σ-restricted $\psi \in \mathsf{UCRPQ}^{=}$, one can decide in polynomial time whether $[\![\psi]\!](\mathsf{G}^{\varepsilon}_{\mathsf{mp}}) \neq \emptyset$. The latter holds as $\mathsf{G}^{\varepsilon}_{\mathsf{mp}}$ contains only a single node, and no edges with labels from Σ. Hence, path variables that occur in string equalities can only be mapped to the empty path, which has label ε. Thus, once can consider each ψ_i from $\psi = \bigvee_{i=1}^{k} \psi_i$ by itself, and observe that $[\![\psi_i]\!](\mathsf{G}^{\varepsilon}_{\mathsf{mp}}) \neq \emptyset$ holds if and only if for each range $L_j(\pi_j)$ of ψ_i, the following holds:

– if π_j occurs in an equality check of ψ_i, then $\varepsilon \in L_j$,
– if π_j does not occur in an equality check of ψ_i, then $L_j \cap \{\triangleright, \triangleleft\}^* \neq \emptyset$.

Clearly, this can be checked in polynomial time. □

But even in Σ-restricted queries, the ranges for variables that do not occur in equality checks can still contain a combination of letters from Σ and the special marker symbols. This is technically cumbersome; and to simplify our reasoning, we first consider queries that further restrict the use of $\triangleright$ and $\triangleleft$.

Definition 7.11 We say that $\varphi \in \mathsf{UCRPQ}^{=}$ over the alphabet $\Delta := \Sigma \cup \{\triangleright, \triangleleft\}$ is *explicitly marked* (or just *explicit*) if for every range L_j in φ, one of $L_j = \{\triangleright\}$, $L_j = \{\triangleleft\}$, or $L_j \subseteq \Sigma^*$ holds.

In other word, explicit queries use the special symbols only to explicitly designate nodes as first or last node of a marked path. In this way, they could also be understood as queries that can use constants for the first and last node of the marked path. Although a query that is explicit is not necessarily Σ-restricted by definition, it is easy to see that it can be straightforwardly made Σ-restricted (as equality checks over $\triangleright$ and $\triangleleft$ can be replaced). Thus, we can view explicit queries as a subclass of Σ-restricted queries.

We are now ready to observe the following connection between equality UCRPQs and SpLog (recall that we defined PC and $\mathsf{PC}_{\mathsf{rx}}$ in Definition 5.5 back in Section 5.2).

Theorem 7.12 *There are polynomial time conversions in both directions*

1. between PC *and explicit* $\mathsf{CRPQ}^{=}$*,*
2. between $\mathsf{PC}_{\mathsf{rx}}$ *and explicit* $\mathsf{CRPQ}^{=}_{\mathsf{rx}}$*.*

Proof. We prove both claims at once: The second is a special case of the first, which we handle by avoiding the use of automata instead of regular expressions (we mention this in the constructions when it is necessary). Although both directions are comparatively straightforward, the transformation to $\mathsf{CRPQ}^{=}$ requires a little more technical attention. We begin with the other direction.

From $\mathsf{CRPQ}^{=}$ *to* SpLog*:* Consider an explicit $\varphi \in \mathsf{CRPQ}^{=}$. As described in the comment after Definition 7.11, we can also assume that φ is Σ-restricted. Let

$$\varphi(\vec{z}_f) = \exists \vec{z}_b \colon \bigwedge_{1\leq i\leq m} (x_i, \pi_i \colon L_i, y_i) \wedge \bigwedge_{1\leq j\leq n} (\xi_j^L = \xi_j^R),$$

and let $X := \mathsf{NVars}(\varphi)$. The main idea in the construction of the $\mathsf{SpLog}(\mathsf{W})$-formula ψ is that each path variable π_i in φ is represented by a SpLog-variable p_i in ψ. In particular, this translates an RPQ $(x, \pi_i \colon L_i, y)$ with $L_i \subseteq \Sigma^*$ into the quantified conjunction $\exists z \colon (\mathsf{W} = x p_i z) \wedge (\mathsf{W} = yz) \wedge \mathsf{C}_{L_i}(p_i)$, and the equality tests $\pi_i = \pi_j$ are directly transformed into word equations $p_i = p_j$.

Following this idea, we construct an intermediate formula χ that realizes φ, but is not yet a prenex conjunction. By applying a straightforward rewriting, we shall then obtain ψ from χ. We now define

$$\chi := \exists \vec{z}_b, p_1, \ldots, p_m \colon \bigwedge_{i=1}^{m} \chi_i \wedge \bigwedge_{j=1}^{n} \eta_j,$$

where each equality check $(\pi_l = \pi_r)$ in φ defines a word equation $\eta_j := (p_l, p_r)$; and for each RPQ $(x_i, \pi_i \colon L_i, y_i)$ in φ, we define χ_i as follows:

– if $L_i = \{\triangleright\}$, then $\chi_i := (x_i = \varepsilon) \wedge (y_i = \varepsilon) \wedge (p_i = \varepsilon)$,
– if $L_i = \{\triangleleft\}$, then $\chi_i := (\mathsf{W} = x_i) \wedge (\mathsf{W} = y_i) \wedge (p_i = \varepsilon)$,
– if $L_i \subseteq \Sigma^*$, then $\chi_i := \exists z \colon (\mathsf{W} = x_i \cdot p_i \cdot z) \wedge (\mathsf{W} = y_i \cdot z) \wedge \mathsf{C}_{L_i}(p_i)$.

If $\varphi \in \mathsf{CRPQ}^{=}_{\mathsf{rx}}$, we can define the constraint C_{L_i} with a regular expression, which ensures that $\psi \in \mathsf{SpLog}_{\mathsf{rx}}$.

Now, χ is only "almost" a prenex conjunction, as it contains some existential quantifiers inside the conjunctions. We now obtain ψ from χ by renaming all variables that are quantified in this way, and moving the quantifiers outside (as in the proof of Lemma 5.6; and observe that this is compatible with Lemma 4.4). Clearly, all this is possible in polynomial time.

From SpLog *to* $\mathsf{CRPQ}^{=}$*:* Assume we are given a $\mathsf{SpLog}(\mathsf{W})$ prenex conjunction

$$\varphi = \exists \vec{x} \colon (\bigwedge_{i=1}^{m} \eta_i \wedge \bigwedge_{j=1}^{n} C_j).$$

Let $X := \big(\bigcup \mathsf{var}(\eta_i)\big) - \{\mathsf{W}\}$. To simplify our definition, we assume the following:

– All right sides of the word equations η_i are of the same length ℓ (this is not essential, but streamlines the notation). More formally, we assume that each η_i is of the form $\eta_i = (\mathsf{W} = \eta_{i,1} \cdots \eta_{i,\ell})$, with $\eta_{i,j} \in (X \cup \Sigma)$.
– For every variable $x \in X$, there is exactly one constraint C_x in φ.

The second assumption can be ensured by rewriting φ in polynomial time: If there is an $x \in X$ with no constraint, we add the constraint $\mathsf{C}_{\Sigma^*}(x)$. If x has multiple constraints $C_1, \dots, C_k$ with $k \geq 2$, we cannot simply combine these into a single constraint for the intersection, as we would face a blowup that is exponential in k. Instead, we proceed as follows: First, we introduce new existentially quantified variables $\hat{x}_2, \dots, \hat{x}_k, y, z$. We then add the conjunction $(\mathsf{W} = yxz) \land \bigwedge_{2 \leq i \leq k} (\mathsf{W} = y\hat{x}_i z)$, which ensures that every solution maps x and all $\hat{x}_i$ to the same values. Finally, in each C_i with $i \geq 2$, we replace x with $\hat{x}_i$.

Now, let $\vec{x}_f$ be a tuple that contains exactly the variables from $\{x^{\vdash}, x^{\dashv} \mid x \in (X \cap \mathsf{free}(\varphi))\}$, and let $\vec{x}_b$ be a tuple of the variables of X that do not occur in $\vec{x}_f$. We then define the explicit and Σ-restricted $\psi \in \mathsf{CRPQ}^=$ as follows:

$$\psi(\vec{x}_f) := \exists \vec{x}_b, y_{1,0}, \dots, y_{m,\ell} \colon \bigwedge_{x \in X} (x^{\vdash}, \pi_x \colon \Sigma^*, x^{\dashv}) \land \bigwedge_{\substack{x \in X, \\ \eta_{i,j} = x}} (\pi_x = \eta_{i,j})$$
$$\land \bigwedge_{i=1}^{m} \Big((y_{i,0}, \rhd, y_{i,0}) \land (y_{i,\ell}, \lhd, y_{i,\ell}) \land \bigwedge_{j=1}^{\ell} \big((y_{i,j-1}, \rho_{i,j} \colon L_{i,j}, y_{i,j}) \big) \Big)$$

where the languages $L_{i,j}$ are defined as follows:

– if $\eta_{i,j} \in \Sigma$, then $L_{i,j} := \{\eta_{i,j}\}$,
– if $\eta_{i,j} \in X$ with $\eta_{i,j} = x$, let $L_{i,j}$ be the language of the constraint C_x (recall that we ensured above that this is uniquely defined).

In the second case, if $\varphi \in \mathsf{SpLog}_{\mathsf{rx}}$, then we can also ensure that $L_{i,j}$ is defined with a regular expression (if our goal is a $\mathsf{SpLog}_{\mathsf{rx}}$-formula).

In order to understand this construction, first note that for each $x \in X$, the RPQ $(x^{\vdash}, \pi_x \colon \Sigma^*, x^{\dashv})$ defines a path π_x from $x^{\vdash}$ to $x^{\dashv}$. This models all possible substitutions for x in the input word of φ.

The second part of ψ, in the lower row, expresses each word equation η_i as a path from $y_{i,0}$ to $y_{i,l}$, using the markers $\rhd$ and $\rhd$ to ensure that the whole word is matched. Each position $\eta_{i,j}$ of η_i is represented by a path variable $\rho_{i,j}$, and the choice of the range ensures that the constraints are respected. Furthermore, the equalities $\pi_x = \eta_{i,j}$ guarantee that all occurrences of a variable x are replaced in the same way. Like for the other direction, it is clear that the transformation is possible in polynomial time. □

As UCRPQs are disjunctions of CRPQs, and as DPC consists of disjunctions of PC-formulas (again, recall Definition 5.5), we can directly conclude the following.

Corollary 7.13 *There are polynomial time conversions in both directions*

1. *between* DPC *and explicit* $\mathsf{UCRPQ}^=$*,*

2. *between* DPCrx *and explicit* $\mathsf{UCRPQ}^{=}_{\mathsf{rx}}$.

We discuss a significant consequence of Theorem 7.12 in Section 7.4. Before that, we consider the transformation of queries that are not explicitly marked.

Theorem 7.14 *There are polynomial time conversions*

1. *from Σ-restricted* $\mathsf{UCRPQ}^{=}$ *to* SpLog*,*
2. *from Σ-restricted* $\mathsf{UCRPQ}^{=}_{\mathsf{rx}}$ *to* $\mathsf{SpLog}_{\mathsf{rx}}$*.*

Proof. Following the same reasoning as for Corollary 7.13, it suffices to give a construction for conjunctive queries. As in the proof of Theorem 7.12, we treat $\mathsf{CRPQ}^{=}_{\mathsf{rx}}$ as a special case that is mentioned only when necessary.

The main idea of this construction is that we rewrite those underlying RPQs where the range contains words with $\triangleright$ or $\triangleleft$. For each of these queries, we distinguish whether the special symbol is actually used or not. To simplify our construction, we abuse the notation, and allow alternations between conjunctions and disjunctions inside the rewritten queries. This is not a problem, as we can extend the proof of Theorem 7.12 to directly translate these disjunctions into SpLog-disjunctions. In order to define the rewriting, we use the following operations on languages $L \subseteq \Delta^*$ with $a \in \{\triangleright, \triangleleft\}$:

– the *a-elimination*, $\mathsf{elim}_a(L) := L \cap (\Delta - \{a\})^*$,
– the *left quotient by* a^+, $\mathsf{lq}_{a^+}(L) := \{v \mid uv \in L \text{ for some } u \in \{a\}^+\}$,
– the *right quotient by* a^+, $\mathsf{rq}_{a^+}(L) := \{u \mid uv \in L \text{ for some } v \in \{a\}^+\}$

We shall discuss further down how these operations can be implemented efficiently on NFAs and regular expressions. Before that, we discuss the rewriting.

Consider any RPQ $(x, \pi\colon L, y)$ that is part of the input Σ-restricted equality CRPQ. Assume that L contains a word in which $\triangleright$ occurs. We now rewrite this RPQ into the following disjunction of equality CRPQs:

$$(x, \pi\colon \mathsf{elim}_{\triangleright}(L), y) \vee \big((x, \triangleright, x) \wedge (x, \pi\colon \mathsf{elim}_{\triangleright}(\mathsf{lq}_{\triangleright^+}(L)), y)\big)$$

In the left part of the disjunction, we handle the case that $\triangleright$ is not used. The resulting language might be empty, but this is not a problem. In the right part, we first express that $\triangleright$ is read at least once. This is done by using $(x, \triangleright, x)$, and allows us to restrict the path π to $\mathsf{lq}_{\triangleright^+}(L)$. But as we can consume arbitrarily many $\triangleright$ this way, we can also restrict π to use only letters from $\Delta - \{\triangleright\}$. As our input query is Σ-restricted, we know that π does not occur in equality checks. Hence, this rewriting does not change the behaviour on marked paths.

After this rewriting, we can assume that all ranges consist of subsets of $(\Sigma \cup \{\triangleleft\})^*$. We now consider all underlying RPQs $(x, \pi\colon L, y)$ where L contains words with $\triangleleft$. These are rewritten into

$$(x, \pi\colon \mathsf{elim}_{\triangleleft}(L), y) \wedge \big((y, \triangleleft, y) \wedge (x, \pi\colon \mathsf{elim}_{\triangleleft}(\mathsf{rq}_{\triangleleft^+}(L)), y)\big)$$

The reasoning is exactly the same as in the case for $\triangleright$. These rewriting steps results in ranges that are $\{\triangleright\}$, $\{\triangleleft\}$, or a subset of Σ^*. Hence, if we "multiplied out" the constructed query, we would obtain an explicit Σ-restricted equality

UCRPQ that, on marked paths, is equivalent to the input query. Obviously, this might result in a query of exponential size. But if we instead allow the transformation from the proof of Theorem 7.12 to convert the path query disjunctions directly into SpLog-disjunctions, we obtain a polynomial time transformation to SpLog, assuming that we can guarantee (as promised above) that the language operations can be computed in polynomial time.

If a range is defined using an NFA A, this can be shown by combining some standard constructions (which can be found in e. g. Hoproft and Ullman [27]): For a-eliminiation, we construct an NFA for $\mathsf{elim}_a(\mathcal{L}(A))$ by removing all transitions with the label a from the NFA A. This is clearly possible in polynomial time. For left quotient by a^+, we first convert A into an NFA with multiple initial states A', where the initial states of A' are those states of A that can be reached by using only transitions with label a. We then convert A' into an equivalent NFA. Again, each of these steps is possible in polynomial time. Finally, we observe that the right quotient by a^+ can be computed in polynomial time by using the left quotient by a^+ and the reversal operation.

If the range is defined with a regular expression α, we first observe that a regular expression for the a-elimination can be computed by replacing all occurrences of a in α with $\emptyset$. The only complicated case is the left quotient by a^+. Luckily, the main result of Gruber and Holzer [25] states that there exists a regular expression α' for this language, and that α' is of polynomial size. Moreover, the proof in [25] also implies that α' can be computed in polynomial time. Finally, the reversal of a regular expression can be computed by reversing the expression, which allows us to reduce the right quotient by a^+ to the left quotient, as we did in the automata case. $\square$

The author considers it unlikely that the other conversion direction is possible in polynomial time: The SpLog-formulas that are derived from the construction in the proof of Theorem 7.14 use disjunctions in a very restricted way (as both parts of the disjunction look “rather similar”), and alternate a bounded number of times between disjunctions and conjunctions. A polynomial time transformation in the opposite direction would need to handle disjunctions of arbitrary formulas, and arbitrary numbers of alternations.

7.4 Adapting Undecidability Results for ECPRQs to Spanners

We conclude our comparison of equality UCRPQs and SpLog with a brief discussion on how Theorem 7.12 can be used to refine some results of Freydenberger and Holldack [16]. As shown in Theorem 4.6 of [16], it is undecidable whether a core spanner representation defines a regular spanner (in other words, whether string equality selections $\zeta^=$ are necessary to define the spanner). This holds even for spanners from the fragment $\mathsf{RGX}^{\{\pi,\zeta^=,\cup\}}$ (i. e., $\mathsf{RGX}^{\mathsf{core}}$ without $\bowtie$).

More importantly, this also affects the relative succinctness of core and regular spanner representations, and the complementation of core spanners: In both cases, the transformation can lead to blowups that are not bounded by any recursive function (see Theorems 4.9 to 4.11 of [16]). By Theorem 4.9, these results also apply to SpLog-formulas. In particular, the trade-off from SpLog to regular languages (regardless of whether they are defined by regular expressions or by NFAs) is not bounded by any recursive function.

Similar results were obtained for ECRPQs by Freydenberger and Schweikardt in [19] (see in particular Theorem 4.6 in that paper). Notably, the proofs of most results in [19] do not require the full power of ECRPQs, but use equality CRPQs instead. Moreover, most of the proofs are restricted to arguing on graphs that are paths, and can directly be used for $\mathsf{CRPQ}^{=}$ on marked paths (see in particular the definition of F_L in Section 4 of [19]). The proof of Theorem 4.6 in [19] fits the second of these criteria, and we can directly transform its query into an explicit Σ-restricted query; the only (minor) problem is that it also uses not just string equalities, but also the special k-ary relation $R_{\mathsf{xor}} := \{(w_1, \ldots, w_k) \mid \text{there is exactly one } i \text{ with } w_i \neq \varepsilon\}$.

Luckily, the constructed query applies R_{xor} only to variables $c^{\bigstar}_{1,1}, \ldots, c^{\bigstar}_{k,1}$, each having the range $\{\varepsilon, \bigstar\}$. Thus, we can express this specific use of R_{xor} as

$$(\mathsf{W} = x \cdot c^{\bigstar}_{1,1} \cdots c^{\bigstar}_{k,1} \cdot y) \land (\mathsf{W} = x \cdot \bigstar \cdot y) \land \bigwedge_{i=1}^{k} \mathsf{C}_{(\varepsilon \lor \bigstar)}(c^{\bigstar}_{i,1}),$$

or, alternatively, its $\mathsf{CRPQ}^{=}$-equivalent. Hence, we can combine these observations, the proof of Theorem 4.6 in [19], and our Theorem 7.12 to conclude that the aforementioned undecidabilities and non-recursive blowups still hold if we do not consider all of SpLog, but only prenex conjunctions. Finally, note that the transformation of SpLog to spanner representations from the proof of Theorem 4.9 (see Section 4.2.1) transforms $\mathsf{PC}_{\mathsf{rx}}$ to core spanner representations from the fragment $\mathsf{RGX}^{\{\pi,\zeta^{=},\times\}}$; and the analogous result holds for PC and $\mathsf{VA}^{\{\pi,\zeta^{=},\times\}}$. Thus, while [16] showed that these results hold for core spanners without join, we can also conclude that they hold for core spanners that do not use union, and only use join as cross product.

8 Negation for SpLog and Difference for Spanners

Fagin et al. [13] also examined core spanners that are extended with a difference operator. Let P_1 and P_2 be spanners with $\mathsf{SVars}(P_1) = \mathsf{SVars}(P_2)$. Then their *difference* $P_1 - P_2$ is defined by $\mathsf{SVars}(P_1 - P_2) := \mathsf{SVars}(P_1)$ and $(P_1 - P_2)(w) = P_1(w) - P_2(w)$ for all $w \in \Sigma^*$. As shown in [13], $[\![\mathsf{RGX}^{\mathsf{core}\cup\{-\}}]\!] \supset [\![\mathsf{RGX}^{\mathsf{core}}]\!]$. In other words, allowing the difference operator increases the expressive power of core spanners.

As one of the reviewers pointed out, this raises the question whether the strong connection between SpLog and core spanners also exists between SpLog with negation and core spanners with a difference operator. First, note that Quine [38] observed already as far back as 1942 that extending EC with negation results in an undecidable theory[9]. More specifically, satisfiability and evaluation both become undecidable. In order to define negation for SpLog, we basically have two choices: One is starting with a definition of $\mathsf{EC}^{\mathsf{reg}}$ that is extended with negation, and then restricting the syntax to SpLog with negation. The other is directly defining syntax and semantics of SpLog that is extended with negation (while ignoring negation for $\mathsf{EC}^{\mathsf{reg}}$). In order to keep the formulas cleaner, we shall choose the second approach; but this does not affect the results that we obtain.

We now define SpLog with negation, or $\mathsf{SpLog}^{\neg}$ for short.

Definition 8.1 Let $\mathsf{W} \in \Xi$. Then $\mathsf{SpLog}^{\neg}(\mathsf{W})$, the set of all $\mathsf{SpLog}^{\neg}$-*formulas with main variable* W, is defined by extending the recursive definition of SpLog from Definition 4.2 with the additional rule that if $\varphi \in \mathsf{SpLog}^{\neg}(\mathsf{W})$, then $(\neg\varphi) \in \mathsf{SpLog}^{\neg}(\mathsf{W})$, with $\mathsf{free}(\neg\varphi) = \mathsf{free}(\varphi)$. We define $\mathsf{SpLog}^{\neg}_{\mathsf{rx}}$ analogously to $\mathsf{SpLog}_{\mathsf{rx}}$.

The semantics of $\mathsf{SpLog}^{\neg}$ extend the semantics of SpLog by defining that $\sigma \models \neg\varphi$ if we have

1. $\sigma(x) \sqsubseteq \sigma(\mathsf{W})$ for all $x \in \mathsf{free}(\varphi)$, and
2. $\sigma \models \varphi$ does not hold.

We apply all notational conventions for $\mathsf{EC}^{\mathsf{reg}}$ and SpLog to $\mathsf{SpLog}^{\neg}$ as well. Regarding the definition of the semantics of negation, note that the first condition ("all free variables need to map to subwords of the main variable") is used to ensure that $\mathsf{SpLog}^{\neg}$ behaves like SpLog in the sense that it guarantees that all variables are safe. We could achieve the same behavior syntactically if we dropped that condition in the semantics, and required that negation is only used in formulas that are guarded, in a manner like $\neg\varphi \wedge \bigwedge_{x \in \mathsf{free}(\varphi)-\{\mathsf{W}\}} x \sqsubseteq \mathsf{W}$. This would shift the effort of ensuring safety from the semantics to the syntax, and result in less readable formulas. As stated above, this would not affect the results in this section.

The notions of formulas that realize spanners, and vice versa, that are given in Definition 4.6 and Definition 4.7 (respectively) directly generalize from SpLog to $\mathsf{SpLog}^{\neg}$. Building on these, we observe the following.

Lemma 8.2 *Let $\varphi_1, \varphi_2 \in \mathsf{SpLog}^{\neg}(\mathsf{W})$ be formulas that realizes spanners P_1 and P_2, respectively. Then $\varphi_1 \wedge \neg\varphi_2$ realizes $P_1 - P_2$.*

Proof. This follows directly from Definition 4.6, extended to $\mathsf{SpLog}^{\neg}$. □

The other direction requires more technical effort. This is due to a peculiar aspect of Definition 4.7 (also recall the discussion after it), which is explained in more detail in the proof of the following rather technical result.

[9] This description of Quine [38] is based on the notations that are used in the current paper. The actual order of events was that first Quine examined the theory of concatenation; and EC was later introduced as its existential positive fragment.

Lemma 8.3 *Let $\varphi \in \mathsf{SpLog}^{\neg}(\mathsf{W})$, and let P be a spanner that realizes φ. Let $X := \mathsf{SVars}(P)$ and define $\Upsilon_X := \bowtie_{x \in X} [\![\Sigma^* x\{\Sigma^*\}\Sigma^*]\!]$. We use $\hat{P}$ to denote the spanner that is obtained from P by renaming each variable $x \in X$ into a new variable $\hat{x}$, and define $P_{\neg} := \Upsilon_X - \pi_X S\big(\Upsilon_X \times \hat{P}\big)$, where S is a sequence of string equality selections that contains exactly the selections $\zeta^{=}_{x,\hat{x}}$ with $x \in X$. Then $P_{\neg}$ realizes $\neg\varphi$.*

Proof. First, note that Υ_X is the universal spanner for X. See [13] for details; for our purposes, it suffices to know that for all $w \in \Sigma^*$, we have that $\Upsilon_X(w)$ contains all possible (X, w)-tuples.

Next, recall that according to Definition 4.7 we have $\sigma \in [\![\varphi]\!](w)$ if and only if there exists *some* $\mu \in P(w)$ with $w_{\mu(x)} = \sigma(x)$ for all $x \in \mathsf{SVars}(P)$. But this is not enough implement negation through the difference operator. For this, we need to describe *all* such (X, w)-tuples. This issue was already mentioned in the discussion after Definition 4.7, and $P' := \pi_X S\big(\Upsilon_X \bowtie \hat{P}\big)$ is the result of the construction that is described there. By definition, $\mu \in P'(w)$ holds if and only if there is some $\hat{\mu} \in \hat{P}(w)$ with $w_{\mu(x)} = w_{\hat{\mu}(\hat{x})}$ for all $x \in X$. As $\hat{P}$ is just a renamed version of P, and as P realizes φ, we conclude $\mu \in P'(w)$ if and only if $\sigma_\mu \models \varphi$, where the substitution σ_μ is defined by $\sigma_\mu(\mathsf{W}) := w$ and $\sigma_\mu(x) := w_{\mu(x)}$ for all $x \in X$.

Finally, observe that for every $w \in \Sigma^*$, we have $\mu \in P_{\neg}(w)$ if and only if μ is an (X, w)-tuple and $\mu \notin P'(w)$, which (according to the previous paragraph) holds if and only if μ is an (X, w)-tuple but we do not have $\sigma_\mu \models \varphi$. In other words, $P_{\neg}$ realizes $\neg\varphi$. $\square$

By adding Lemma 8.2 and Lemma 8.3 to the proof of Theorem 4.9, we can directly extend the latter to cover $\mathsf{SpLog}^{\neg}$. The same applies to Corollary 4.10. We summarize this as follows.

Theorem 8.4 *There are polynomial time conversions*

1. *from $\mathsf{RGX}^{\mathsf{core}\cup\{-\}}$ to $\mathsf{SpLog}^{\neg}_{\mathsf{rx}}$, and from $\mathsf{SpLog}^{\neg}_{\mathsf{rx}}$ to $\mathsf{RGX}^{\mathsf{core}\cup\{-\}}$,*
2. *from $\mathsf{SpLog}^{\neg}$ to $\mathsf{VA}^{\mathsf{core}\cup\{-\}}_{\mathsf{set}}$ and to $\mathsf{VA}^{\mathsf{core}\cup\{-\}}_{\mathsf{stk}}$,*
3. *modulo ε from $\mathsf{VA}^{\mathsf{core}\cup\{-\}}$ to $\mathsf{SpLog}^{\neg}$.*

There are polynomial size conversions from $\mathsf{VA}^{\mathsf{core}\cup\{-\}}$ to $\mathsf{SpLog}^{\neg}$. These conversions run in polynomial time if all v-automata in the spanner representation are functional.

In other words, the relation between SpLog and core spanners is the same as the one between $\mathsf{SpLog}^{\neg}$ and core spanners with difference. Likewise, $\mathsf{SpLog}^{\neg}$ can be used to define relations for core spanners with difference. Hence, $\mathsf{SpLog}^{\neg}$ and core spanners with difference can be used as interchangeably as SpLog and core spanners. A thorough study of the the properties of $\mathsf{SpLog}^{\neg}$ (and, thereby, core spanners with difference) is outside the scope of the current paper, and left to future publication.

We only note that some properties of $\mathsf{SpLog}^{\neg}$ follow immediately from previously known properties of core spanners or the theory of concatenation.

For example, we can directly conclude from the undecidability of core spanner universality (see [16]) that satisfiability is undecidable for $\mathsf{SpLog}^\neg$ (this also follows, although with a little more effort, from the fact that the theory of concatenation is undecidable, see [38]). Another direct consequence of [16] is that the blowup from $\mathsf{SpLog}^\neg$ to SpLog is not bounded by any recursive function. Of course, like one of the reviewers pointed out, the restriction that each variable is mapped to a subword of the main variable ensures that evaluation of $\mathsf{SpLog}^\neg$ is satisfiable; and it is easily seen that for inputs $\varphi \in \mathsf{SpLog}^\neg$ and σ, one can decide in PSPACE whether $\sigma \models \varphi$, by reasoning analogously to the NP upper bound in Corollary 4.12.

In contrast to these easy pickings, there is no general inexpressibility result for $\mathsf{SpLog}^\neg$ (like Theorem 6.5 for SpLog). The author is not aware of a result for EC with negation that corresponds to Theorem 6.4. While some pumping result for $\mathsf{SpLog}^\neg$ (and, hence, core spanners with difference) would be very interesting, it seems safe to assume that finding such a result is even more challenging than finding new inexpressibility results for SpLog. But at the very least, $\mathsf{SpLog}^\neg$ provides us with an alternative approach to examining the expressive power of core spanners with difference.

9 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. [13], and it is possible to convert between these models in polynomial time (and the analogous result holds for $\mathsf{SpLog}^\neg$ and core spanners with difference). 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. the formula $\varphi_{\neq}$ in Example 5.3). Of course, whether one considers SpLog or one of the spanner representations more convenient 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 or extended in a suitable way.

Another set of question where the comparatively simple syntax and semantics of SpLog might prove useful 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 $\mathsf{SpLog}_{\mathsf{rx}}$. The author conjectures that this blowup is exponential.

As another topic, note that 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 4.11, 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 [39].

A more fundamental question is whether $[\![\mathsf{EC}^{\mathsf{reg}}]\!] = [\![\mathsf{SpLog}]\!]$. In addition to our discussion in Section 5.2, a potential approach to this is examining whether every bounded $\mathsf{EC}^{\mathsf{reg}}$-language is an EC-language (as the reasoning from Theorem 6.2 does not carry over from SpLog to $\mathsf{EC}^{\mathsf{reg}}$). As a related question, the expressive power of $\mathsf{SpLog}^{\neg}$ remains open (aside of $[\![\mathsf{SpLog}]\!] \subset [\![\mathsf{SpLog}^{\neg}]\!]$, which follows from [13]).

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 a friendlier upper bound than evaluation of $\mathsf{EC}^{\mathsf{reg}}$ (NP and PSPACE, respectively). While we have only defined SpLog with a single main variable, a natural generalization would be allowing multiple main variables (the definition generalizes naturally to "every variable is a subword of one of the main variables", and the upper bound for evaluation remains). A potential application of SpLog with two multiple variables is describing relations for path labels in graph databases.

Acknowledgements The author thanks Wim Martens for helpful comments and discussions, Benny Kimelfeld for answering questions, and the anonymous reviewers of this and the conference version for their insightful feedback and suggestions.

References

1. Abigail. Re: Random number in perl. Posting in the newsgroup comp.lang.perl.misc, October 1997. Message-ID slrn64sudh.qp.abigail@betelgeuse.wayne.fnx.com.
2. Pablo Barceló, Leonid Libkin, Anthony Widjaja Lin, and Peter T. Wood. Expressive languages for path queries over graph-structured data. *ACM Transactions on Database Systems*, 37(4):31:1–31:46, 2012.
3. Pablo Barceló and Pablo Muñoz. Graph logics with rational relations: The role of word combinatorics. *ACM Transactions on Computational Logic*, 18(2):10:1–10:41, 2017.
4. Jean-Camille Birget. Intersection and union of regular languages and state complexity. *Information Processing Letters*, 43(4):185–190, 1992.
5. Benjamin Carle and Paliath Narendran. On extended regular expressions. In *LATA 2009*, 2009.

6. Christian Choffrut and Juhani Karhumäki. Combinatorics of words. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 1: Word, Language, Grammar, chapter 6, pages 329–438. Springer, 1997.
7. Laura Ciobanu, Volker Diekert, and Murray Elder. Solution sets for equations over free groups are EDT0L languages. In *ICALP 2015*, 2015.
8. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, 3rd edition, 2009.
9. Elena Czeizler. The non-parametrizability of the word equation $xyz{=}zvx$: A short proof. *Theoretical Computer Science*, 345(2-3):296–303, 2005.
10. Volker Diekert. Makanin's Algorithm. In M. Lothaire, editor, *Algebraic Combinatorics on Words*, chapter 12. Cambridge University Press, 2002.
11. Volker Diekert. More than 1700 years of word equations. In *CAI 2015*, 2015.
12. Andrzej Ehrenfeucht and Grzegorz Rozenberg. A pumping theorem for EDT0L languages. Technical report, Tech. Rep. CU-CS-047-74, University of Colorado, 1974.
13. Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. Document spanners: A formal approach to information extraction. *Journal of the ACM*, 62(2):12, 2015.
14. Dominik D. Freydenberger. Extended regular expressions: Succinctness and decidability. *Theory of Computing Systems*, 53(2):159–193, 2013.
15. Dominik D. Freydenberger. A logic for document spanners. In *ICDT 2017*, 2017.
16. Dominik D. Freydenberger and Mario Holldack. Document spanners: From expressive power to decision problems. *Theory of Computing Systems*, 62:854—-898, 2018.
17. Dominik D. Freydenberger, Benny Kimelfeld, and Liat Peterfreund. Joining extractions of regular expressions. In *PODS 2018*, 2018.
18. Dominik D. Freydenberger and Markus L. Schmid. Deterministic regular expressions with back-references. In *STACS 2017*, 2017.
19. Dominik D. Freydenberger and Nicole Schweikardt. Expressiveness and static analysis of extended conjunctive regular path queries. *Journal of Computer and System Sciences*, 79(6):892–909, 2013.
20. Zvi Galil. Hierarchies of complete problems. *Acta Informatica*, 6:77–88, 1976.
21. Michael R. Garey and David S. Johnson. *Computers and Intractability.* W. H. Freeman and Company, 1979.
22. Seymour Ginsburg. *The Mathematical Theory of Context-Free Languages.* McGraw-Hill, 1966.
23. Seymour Ginsburg and Edwin H. Spanier. Bounded regular sets. *Proceedings of the American Mathematical Society*, 17(5):1043–1049, 1966.
24. Sheila A. Greibach. A note on undecidable properties of formal languages. *Mathematical Systems Theory*, 2(1):1–6, 1968.
25. Hermann Gruber and Markus Holzer. Language operations with regular expressions of polynomial size. *Theoretical Computer Science*, 410(35):3281–3289, 2009.
26. Hermann Gruber and Markus Holzer. From finite automata to regular expressions and back - A summary on descriptional complexity. *International Journal of Foundations of Computer Science*, 26(8):1009–1040, 2015.
27. John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.
28. Lucian Ilie. Subwords and power-free words are not expressible by word equations. *Fundamenta Informaticae*, 38(1-2):109–118, 1999.
29. Lucian Ilie and Wojciech Plandowski. Two-variable word equations. *RAIRO–Theoretical Informatics and Applications*, 34(6):467–501, 2000.
30. Juhani Karhumäki, Filippo Mignosi, and Wojciech Plandowski. The expressibility of languages and relations by word equations. *Journal of the ACM*, 47(3):483–505, 2000.
31. Juhani Karhumäki, Wojciech Plandowski, and Wojciech Rytter. Generalized factorizations of words and their algorithmic properties. *Theoretical Computer Science*, 218(1):123–133, 1999.
32. Juhani Karhumäki and Aleksi Saarela. An analysis and a reproof of Hmelevskii's theorem. In *DLT 2008*, 2008.
33. Lila Kari, Grzegorz Rozenberg, and Arto Salomaa. L systems. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 1: Word, Language, Grammar, chapter 5, pages 253–328. Springer-Verlag New York, 1997.

34. Yunyao Li, Frederick Reiss, and Laura Chiticariu. SystemT: A declarative information extraction system. In *ACL 2011*, 2011.
35. Wim Martens and Frank Neven. Frontiers of tractability for typechecking simple XML transformations. *Journal of Computer and System Sciences*, 73(3):362–390, 2007.
36. Alexandru Mateescu and Arto Salomaa. Aspects of classical language theory. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 1: Word, Language, Grammar, chapter 4, pages 175–251. Springer, 1997.
37. Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.
38. W. V. Quine. Concatenation as a basis for arithmetic. *The Journal of Symbolic Logic*, 11(4):105–114, 1946.
39. Daniel Reidenbach and Markus L. Schmid. Patterns with bounded treewidth. *Information and Computation*, 239:87–99, 2014.
40. Thomas J. Schaefer. The complexity of satisfiability problems. In *STOC 1978*, 1978.
41. Markus L. Schmid. Characterising REGEX languages by regular languages equipped with factor-referencing. *Information and Computation*, 249:1–17, 2016.
42. Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley, 2011.
43. Larry J. Stockmeyer and Albert R. Meyer. Word problems requiring exponential time: Preliminary report. In *STOC 1973*, 1973.