Regular expressions for data words

Libkin, Leonid; TAN, Tony & Vrgoc, Domagoj (2015) Regular expressions for data words. In: JOURNAL OF COMPUTER AND SYSTEM SCIENCES, 81(7), p. 1278-1297.

DOI: 10.1016/j.jcss.2015.03.005
Handle: http://hdl.handle.net/1942/18447

Regular languages for data words

Leonid Libkin

University of Edinburgh

Tony Tan

Hasselt University and Transnational University of Limburg

Domagoj Vrgoč

University of Edinburgh

Abstract

In data words, each position carries not only a letter form a finite alphabet, as the usual words do, but also a data value coming from an infinite domain. There has been a renewed interest in them due to applications in querying and reasoning about data models with complex structural properties, notably XML, and more recently, graph databases. Logical formalisms designed for querying such data often require concise and easily understandable presentations of regular languages over data words.

Our goal, therefore, is to define and study regular expressions for data words. As the automaton model, we take register automata, which are a natural analog of NFAs for data words. We first define *regular expression with memory* (REM), which extend standard regular expressions with limited memory, and show that they capture the class of data words defined by register automata. The complexity of the main decision problems for REM also turns out to be the same as for register automata.

In order to lower the complexity of main reasoning tasks we then look at two natural subclasses of REM that can define many properties of interest in applications of data words: regular expression with binding (REWB) and regular expression with equality (REWE). We analyse both of these classes in terms of computational complexity and establish a strict hierarchy of fragments in terms of their expressive power.

Keywords: Data words, register automata, regular expressions

1. Introduction

Data words are words that, in addition to a letter from a finite alphabet, have a *data value* from an infinite domain associated with each position. For example, $\binom{a}{1}\binom{b}{2}\binom{b}{1}$ is a data word over the alphabet $\Sigma = \{a, b\}$ and $\mathbb{N}$ as the domain of values. It can be viewed as the ordinary word abb in which the first and the third positions are equipped with value 1, and the second position with value 2.

These were introduced in [1] which proposed a natural extension of finite automata for them, called *register automata*. Data words have become an active subject of research lately due to their applications in XML, in particular in static analysis of logic and automata-based XML specifications, and in query evaluation tasks. Indeed, paths in XML trees should account not only for the labels (XML tags) but values of attributes, which can come from an infinite domain, such as $\mathbb{N}$. While logic and automata models are well-understood by now for the structural part of XML (i.e., trees) [2, 3, 4], adding data values required a concentrated effort for finding good logics and their associated automata [5, 6, 7, 8, 9, 10]. Connections between logical and automata formalisms have been explored as well, usually with the focus on finding logics with decidable satisfiability problem. A well-known result of [7] shows that FO^2, the two-variable fragment of first-order logic extended by equality test for data values, is decidable over data words. Another account of this was given in [9], where various data word automata models are compared to fragments of FO and MSO with regard to their expressive power. Recently, the problem was studied in [11, 12]; in particular it was shown that the guarded fragment of MSO defines data word languages that are recognized by non-deterministic register automata.

Data words appear in other areas as well, in particular verification, and querying databases. In several applications, one would like to deal with concise and easy-to-understand representations of languages of data words. These can be used, for example, in extending languages for XML navigation that take into account data values. Another possible example is in the field of verification, in particular from modeling infinite-state systems with finite control [13, 14]. Here having a concise representation of system properties is much preferred to long and unintuitive specifications given by e.g. automata.

The need for a good representation mechanism for data word languages is particularly apparent in the area of querying graph databases [15], a data model that is increasingly common in applications including social networks, biology, Semantic Web, and RDF. Many properties of interest in such databases are expressed by regular path queries [16], asking for the existence of a path conforming to a given

regular expression, or their extensions [17, 18]. Typical queries are specified by the closure of atomic formulae $x \xrightarrow{L} y$ under $\wedge$ and $\exists$; the atoms ask for the existence of a path between x and y whose label is in the regular language L [17]. Typically, such logical languages have been studied without taking data values into account. Recently, however, logical languages that extend regular conditions from words to data words appeared [19]; for such languages we need a concise way of representing regular languages, which is most commonly done by regular expressions (as automata tend to be too cumbersome to be used in a query language).

The most natural extension of the usual NFAs to data words is *register automata*, first introduced in [1] and studied, for example, in [13, 20]. These are in essence finite state automata equipped with a set of registers that allow them to store data values and make a decision about their next step based not only on the current state and the letter in the current position, but also by comparing the current data value with the ones previously stored in registers. They were originally introduced as a mechanism to reason about words over an infinite alphabet (that is, without the finite part), but they easily extend to describe data word languages. Note that a variety of other automata formalisms for data words exist, for example, pebble automata [9, 21], data automata [7], and class automata [6]. In this paper we concentrate on languages specified by register automata, since they are the most natural generalization of finite state automata to languages over data words.

As mentioned earlier, if we think of a specification of a data word language, register automata are not the most natural way of providing them: in fact, even over the usual words, regular languages are easier to describe by regular expressions than by NFAs. For example, in XML and graph database applications, specifying paths via regular expressions is completely standard. In many XML specifications (e.g., XPath), data value comparisons are fairly limited: for instance, one checks if two paths end with the same value. On the other hand, in graph databases, one often needs to specify a path using both labels and data values that occur in it. For those purposes, we need a language for describing regular languages of data words, i.e., languages accepted by register automata. In [19] we started looking at such expressions, but in a context slightly different from data words. Our goal now is to present a clean account of regular expressions for data words that would:

1. capture the power of register automata over data words, just as the usual regular expressions capture the power of regular languages;

2. have good algorithmic properties, at least matching those of register automata; and
3. admit expressive subclasses with very good (efficient) algorithmic properties.

For this, we define three classes of regular expressions (in the order of decreasing expressive power): regular expressions with memory (REM), regular expressions with binding (REWB), and regular expressions with equality (REWE).

Intuitively, REM are standard regular expression extended with a finite number of variables, which can be used to bind and compare data values. It turns out that REM have the same expressive power as register automata. Note that an attempt to find such regular expressions has been made in [22], but it fell short of even the first goal. In fact, the expressions of [22] are not very intuitive, and they fail to capture some very simple languages like, for example, the language $\{\binom{a}{d}\binom{a}{d'} \mid d \neq d'\}$. In our formalism this language will be described by a regular expression $(a{\downarrow}x) \cdot (a[x^{\neq}])$. This expression says: bind x to be the data value seen while reading a, move to the next position, and check that the symbol is a and that the data value differs from the one in x. The idea of binding is, of course, common in formal language theory, but here we do not bind a letter or a subword (as, for example, in regular expressions with backreferencing) but rather values from an infinite alphabet. This is also reminiscent of freeze quantifiers used in connection with the study of data word languages [13].

However, one may argue that the binding rule in REM may not be intuitive. Consider the following expression: $a \downarrow x(a[x^{=}]a \downarrow x)^* a[x^{=}]$. This expression rebinds variable x inside the scope of another binding, and then crucially, when this happens, the original binding of x is lost. Such expressions really mimic the behavior of register automata, which makes them more procedural than declarative. (The above expression defines data words of the form $\binom{a}{d_1}\binom{a}{d_1} \cdots \binom{a}{d_n}\binom{a}{d_n}$.)

Losing the original binding of a variable when reusing it inside its scope goes completely against the usual practice of writing logical expressions, programs, etc., that have bound variables. Nevertheless, as we show in this paper, this feature was essential for capturing register automata. A natural question then arises about expressions using proper scoping rules, which we call regular expressions with binding (REWB) and show that they are strictly weaker than REM. The non emptiness problem for REWB drops to NP-complete, while the universality remains undecidable.

Finally, we introduce and study regular expressions with equality (REWE). Intuitively, REWE are regular expressions extended with operators $(e)_{=}$ and $(e)_{\neq}$,

which denotes the language of data words which conform to e, where the first and the last data values are the same and different, respectively. We show that REWE is strictly weaker than REWB, but its membership and nonemptiness problems become tractable.

Organisation. In Section 2 we review the basic notations and register automata. We recall a few facts on RA in Section 3. For the sake of completeness, we reprove some of them. In Sections 4, 5 and 6 we introduce and study REM, REWB and REWE, respectively. Finally we end with some concluding remarks in Section 7.

2. Notations and definitions

We recall the definition of data words and formally define the standard decision problems and closure properties.

Data words. A data word over Σ is a finite string over the alphabet $\Sigma \times \mathcal{D}$, where Σ is a finite set of letters and $\mathcal{D}$ an infinite set of data values. That is, in each position a data word carries a letter from Σ and a data value from $\mathcal{D}$. We will write data words as $\binom{a_1}{d_1} \dots \binom{a_n}{d_n}$, where the label and the data value in position i are a_i and d_i, respectively. The set of all data words over the alphabet Σ and set of data values $\mathcal{D}$ is denoted by $(\Sigma \times \mathcal{D})^*$. A data word language is a subset $L \subseteq (\Sigma \times \mathcal{D})^*$. We denote by $\mathsf{Proj}_\Sigma(w)$ the word $a_1 \cdots a_n$, that is, the projection of w to its Σ component.

Register automata. Register automata are an analogue of NFAs for data words. They move from one state to another by reading the appropriate letter from the finite alphabet and comparing the data value to ones previously stored in the registers.

Our version of register automata will use the notion of *conditions* which are boolean combinations of atomic $=, \neq$ comparisons of data values. Formally, conditions are defined as follows. Let $x_1, \dots, x_k$ be variables, denoting the registers. The class of conditions $\mathcal{C}_k$ is defined by the grammar:

$$\varphi \;:=\; \mathtt{tt} \mid \mathtt{ff} \mid x_i^{=} \mid x_i^{\neq} \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi,$$

where $x_i \in \{x_1, \dots, x_k\}$. A condition is *positive*, if it does not contain $x_i^{\neq}$, nor the $\neg$ operator. Intuitively, the condition $x_i^{=}$ ($x_i^{\neq}$) means that the currently read data value is the same (different) as the content of register x_i.

A valuation on the variables $x_1, \dots, x_k$ is a partial function $\nu : \{x_1, \dots, x_k\} \to \mathcal{D}$. We denote by $\mathcal{F}(x_1, \dots, x_k)$ the set of all valuations on

$x_1, \ldots, x_k$. For a valuation ν, we write $\nu[x_i \leftarrow d]$ to denote the valuation ν' obtained by fixing $\nu'(x_i) = d$ and $\nu'(x) = \nu(x)$ for all other $x \neq x_i$. A valuation ν is compatible with a condition $\varphi \in \mathcal{C}_k$, if for every variable x_i that appears in φ, $\nu(x_i)$ is defined.

Let $\nu \in \mathcal{F}(x_1, \ldots, x_k)$ and $d \in \mathcal{D}$. The satisfaction of a condition φ by (d, ν) is defined inductively as follows.

- $d, \nu \models \mathtt{tt}$ and $d, \nu \not\models \mathtt{ff}$.
- $d, \nu \models x_i^{=}$ if and only if $\nu(x_i)$ is defined and $\nu(x_i) = d$.
- $d, \nu \models x_i^{\neq}$ if and only if $\nu(x_i)$ is defined and $\nu(x_i) \neq d$.
- $d, \nu \models \varphi_1 \wedge \varphi_2$ if and only if $d, \nu \models \varphi_1$ and $d, \nu \models \varphi_2$.
- $d, \nu \models \varphi_1 \vee \varphi_2$ if and only if $d, \nu \models \varphi_1$ or $d, \nu \models \varphi_2$.
- $d, \nu \models \neg\varphi$ if and only if $d, \nu \not\models \varphi$.

Definition 2.1 (Register data word automata)**.** *Let Σ be a finite alphabet and k a natural number. A register automaton (RA) with k registers $x_1, \ldots, x_k$ is a tuple $\mathcal{A} = (Q, q_0, F, T)$, where:*

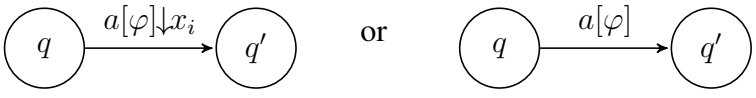
- *Q is a finite set of states;*
- *$q_0 \in Q$ is the initial state;*
- *$F \subseteq Q$ is the set of final states;*
- *T is a finite set of transitions of the form:*

$$q, a[\varphi]{\downarrow}x_i \to q' \qquad \textit{or} \qquad q, a[\varphi] \to q',$$

where q, q' are states, $a \in \Sigma$, $x_i \in \{x_1, \ldots, x_k\}$, and φ is a condition in $\mathcal{C}_k$.

Intuitively on reading the input $\binom{a}{d}$, if the automaton is in state q and there is a transition $q, a[\varphi]{\downarrow}x_i \to q' \in T$ such that $d, \bar{c} \models \varphi$, where $\bar{c} = (c_1, \ldots, c_k)$ is the current content of the registers, then it moves to the state q' while changing the content of register i to d. The transitions $q, a[\varphi] \to q'$ are processed similarly, except that they do not change the content of the registers.

The transitions in RA can be graphically represented as follows:



Let $\mathcal{A}$ be a k-register automaton. A *configuration* of $\mathcal{A}$ on w is a pair $(q, \nu) \in Q \times \mathcal{F}(x_1, \ldots, x_k)$. The initial configuration is (q_0, ν_0), where $\nu_0 = \emptyset$. A configuration (q, ν) with $q \in F$ is a final configuration.

A configuration (q, ν) yields a configuration (q', ν') by $\binom{a}{d}$, denoted by $(q, \nu) \vdash_{a,d} (q', \nu')$, if either

- there is a transition $q, a[\varphi]{\downarrow}x_i \to q'$ of $\mathcal{A}$ such that $d, \nu \models \varphi$ and $\nu' = \nu[x_i \leftarrow d]$, or
- there is a transition $q, a[\varphi] \to q'$ of $\mathcal{A}$ such that $d, \nu \models \varphi$ and $\nu' = \nu$.

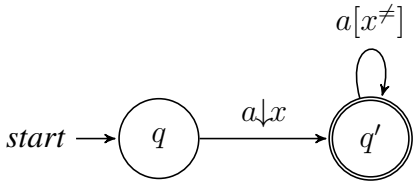
Let $w = \binom{a_1}{d_1} \ldots \binom{a_n}{d_n}$. A *run* of $\mathcal{A}$ on w is a sequence of configurations $(q_0, \nu_0), \ldots, (q_n, \nu_n)$ such that (q_0, ν_0) is the initial configuration and $(q_{i-1}, \nu_{i-1}) \vdash_{a_i,d_i} (q_i, \nu_i)$, for each $i = 1, \ldots, n$. It is called an accepting run if (q_n, ν_n) is a final configuration. We say that $\mathcal{A}$ accepts w, denoted by $w \in L(\mathcal{A})$, if there is an accepting run of $\mathcal{A}$ on w.

For a valuation ν, we define the automaton $\mathcal{A}(\nu)$ that behaves just like the automaton $\mathcal{A}$, but we insist that any run starts with the configuration (q_0, ν), that is, with the content of the registers ν, instead of the empty valuation.

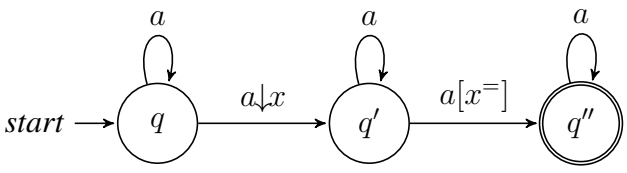
We now present a few examples of register automata.

Example 2.2. *In the following let $\Sigma = \{a\}$. To ease the notation in transitions that have $\varphi = \mathtt{tt}$ we will omit φ and simply write $a{\downarrow}x_i$, or a on the arrows of our automata.*

- *The 1-register automaton $\mathcal{A}_1$ represented below accepts all data words in which the data value in the first position is different from all the other data value.*



- *The 1-register automaton $\mathcal{A}_2$ represented below accepts all data words in which there are two positions with the same data value.*



3. Some useful facts about register automata

In this section we recall the basic language theoretic properties of register data word automata. Most of these results follow from [1], however, since some subtle differences were introduced to the model we will reprove most of the results to make the presentation self contained. Some changes introduced here will have an impact on the nonemptiness problem, as already noted in [20, 13], however, all of the other results remain intact. In order to prove complexity results about membership and nonemptiness we will require some general properties of register automata that we examine next. At the end we will also recall closure properties of the class of languages defined by register automata.

A useful property of register automata that will be needed in what follows is that, intuitively, such automata can only keep track of as many data values as can be stored in their registers. Formally, we have:

Lemma 3.1. *Let $\mathcal{A}$ be a k-register data word automaton. If $\mathcal{A}$ accepts some word of length n, then it accepts a word of length n in which there are at most $k+1$ different data values.*

The proof of Lemma 3.1 can be found in Section 3.1

We now show that we can view register automata as NFAs when restricted only to a finite set of data values.

Let $\mathcal{A} = (Q, q_0, F, T)$ be a k-register data word automaton, D a finite set of data values, and $D_\perp = D \cup \{\perp\}$, with $\perp$ a new symbol used to denote that the register is empty. We transform $\mathcal{A}$ into an NFA $\mathcal{A}_D = (Q', q_0', F', \delta)$ over the alphabet $\Sigma \times D$ as follows:

- $Q' = Q \times D_\perp^k$;
- $q_0' = (q_0, \perp^k)$;
- $F' = F \times D_\perp^k$;
- Whenever we have a transition $q, a[\varphi]{\downarrow}x_i \to q'$ in T, we add the transition

$$((q, \tau), \binom{a}{d}, (q', \tau'))$$

to T if $d, \tau \models \varphi$ and τ' is obtained from τ by putting d in the position i, where both τ and τ' are from $D_\perp^k$.

- Whenever we have a transition $q, a[\varphi] \to q'$ in T, we add the transition

$$((q, \tau), \binom{a}{d}, (q', \tau))$$

to T if $d, \tau \models \varphi$.

It is straightforward to check that $\mathcal{A}$ accepts a data word over $\Sigma \times D$ if and only if $\mathcal{A}_D$ does. That is we obtain the following.

Lemma 3.2. [1, Proposition 1] *Let D be a finite set of data values and $\mathcal{A}$ a k-register RA over Σ. Then there exists a finite state automaton $\mathcal{A}_D$ over the alphabet $\Sigma \times D$ such that $w \in L(\mathcal{A}_D)$ if and only if $w \in L(\mathcal{A})$, for every w with data values from D. Moreover, the number of states in $\mathcal{A}_D$ is $|Q| \cdot (|D| + 1)^k$, where Q is the set of states in $\mathcal{A}$.*

Membership, nonemptiness and universality are some of the most important decision problems related to formal languages.[1] We now recall the exact complexity of these problems for register automata. Since the model of register automata we use here differs slightly from the one in previous work, we sketch how these results carry over to our model.

The following theorem summarises the complexity of RA.

Theorem 3.3.
- *The nonemptiness problem for RA is* PSPACE*-complete* [13].
- *The membership problem for RA is* NP*-complete* [20].
- *The universality and containment problems for RA are undecidable* [9].

Proof sketch. For nonemptiness, the lower bound will follow from Theorem 4.6 and Theorem 4.5. For the upper bound we convert our k-register automaton $\mathcal{A}$ into an NFA $\mathcal{A}_D$ over the alphabet $\Sigma \times D$ (as in the Lemma 3.2), where $D = \{0, \ldots, k+1\}$. We know that $\mathcal{A}_D$ recognizes all data words from $L(\mathcal{A})$ using only data values from D. By Lemma 3.1 and invariance under automorphisms, we know that checking $\mathcal{A}$ for nonemptiness is equivalent to checking $\mathcal{A}_D$ for

[1] To be precise, we present the definitions of the problems here. The *nonemptiness problem* asks us to check, given as input a RA $\mathcal{A}$, whether $L(\mathcal{A}) \neq \emptyset$. The *membership problem* takes as input a RA $\mathcal{A}$ and a data word w, and requires to check whether $w \in L(\mathcal{A})$. The *universality problem* asks, given as input a RA $\mathcal{A}$, to check whether $L(\mathcal{A}) = (\Sigma \times \mathcal{D})^*$. The *containment problem* asks, given as input RAs $\mathcal{A}_1$ and $\mathcal{A}_2$, to check whether $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$.

nonemptiness. Using on-the-fly construction we get the desired result (note that $\mathcal{A}_D$ can not be created before checking it for nonemptiness).

We now move to membership. The lower bound will follow from Theorem 4.6 and Theorem 4.5. For the upper bound it simply suffices to guess an accepting run of the automaton. Since every transition of the automaton processes one symbol of our data word, we only need to guess $|w|$ states of the automaton, where w is the input data word. It is straightforward to check that we can simulate the automaton in PTIME.

The proof of the last item follows directly from [9]. □

However, for positive RA, that is, RA in which the conditions in the transitions are all positive, the containment problem becomes decidable [23, 24].

At this point we should remark that there is a subtle difference between our model here, and the original version introduced by Kaminski and Francez in [1]. In the original version of Kaminski and Francez different registers are not allowed to store the same data value. Such restriction brings the the complexity of nonemptiness problem down to NP-complete [20]. While, in the model here, the non emptiness problem is PSPACE-complete, as stated in the fact above. The reason why we choose this model is that it is essentially the same one used in [13, 10] which has become a more mainstream model of register automata.

Since register automata closely resemble classical finite state automata, it is not surprising that some (although not all) constructions valid for NFAs can be carried over to register automata. We now recall results about closure properties of register automata [1]. Although our notion of automata is slightly different than the one used there, all constructions from [1] can be easily modified to work in the setting proposed here.

Fact 3.4. *[1]*

1. *The set of languages accepted by register automata is closed under union, intersection, concatenation and Kleene star.*
2. *Languages accepted by register automata are not closed under complement.*
3. *Languages accepted by register automata are closed under automorphisms on $\mathcal{D}$: that is, if $f : \mathcal{D} \to \mathcal{D}$ is a bijection and w is accepted by $\mathcal{A}$, then the data word $f(w)$ in which every data value d is replaced by $f(d)$ is also accepted by $\mathcal{A}$.*

3.1. Proof of Lemma 3.1

We first set some notation. The content of the registers will be called an *assignment* and will usually be denoted by τ. That is, for a k-register RA an assignment is simply a partial function from $\{1, \ldots, k\}$ to $\mathcal{D}$. We will say that two k-register assignments τ and $\overline{\tau}$ are of the same equality type if we have $\tau(i) = \tau(j)$ if and only if $\overline{\tau}(i) = \overline{\tau}(j)$, for all $i, j \leq k$. Note that this also implies that $\tau(i) \neq \tau(j)$ if and only if $\overline{\tau}(i) \neq \overline{\tau}(j)$.

We will prove a slightly more general claim, allowing our automata to start with an nonempty assignment of the registers. Let $\mathcal{A}(\tau_0) = (Q, q_0, F, T)$ be a k-register data word automaton, starting with the initial assignment τ_0 in the registers and $w = \binom{a_1}{d_1} \ldots \binom{a_n}{d_n}$ a word that it accepts. This means that there is a sequence of states $q_0, q_1, \ldots, q_n$, with $q_n \in F$ and a sequence of register assignments $\tau_0, \tau_1, \ldots, \tau_n$ such that $(q_{i-1}, \tau_{i-1}) \vdash_{a_i, d_i} (q_i, \tau_i)$, for $i = 1 \ldots n$. Moreover, for each i we have that either $\tau_{i-1} = \tau_i$ (if a transition of the form $q, a[\varphi] \to q'$ is used), or τ_i is obtained from τ_{i-1} by replacing the value in register j by d_i if a transition of the form $q, a[\varphi]{\downarrow}x_j \to q'$ is used.

Now let $\overline{S} = \{\tau_0(i) : 1 \leq i \leq k\} - \{\perp\}$. That is $\overline{S}$ contains all the data values from the initial assignment, except the one denoting that the register is empty.

Let S be any set of data values such that $|S| = k + 1$ and $\overline{S} \subseteq S$.

We prove by induction on $i \leq n$ that we can define a data word w_i, of length i, such that $w_i = \binom{a_1}{d_1^i} \ldots \binom{a_i}{d_i^i}$, where $a_1, \ldots a_i$ are from w and $d_1^i, \ldots, d_i^i$ are from S. We then show that for this w_i there is a sequence of assignments $\tau_0', \tau_1', \ldots \tau_i'$ such that each τ_j' is of the same equality type as τ_j, where $j \leq i$ and it holds that $\tau_{j-1}', d_j^i \models \varphi_j$, for all $j \leq i$, where φ_j is the condition from the transition witnessing that $(q_{j-1}, \tau_{j-1}) \vdash_{a_j, d_j} (q_j, \tau_j)$. Furthermore, each τ_j' is either equal to τ_{j-1}' (if the transition witnessing $(q_{j-1}, \tau_{j-1}) \vdash_{a_j, d_j} (q_j, \tau_j)$ does not use $\downarrow x$), or is obtained from τ_{j-1}' by replacing the data value in position k by d_j (when the transition is of the form $q_{j-1}, a_j[\varphi_j]{\downarrow}x_k \to q_j$). Note that this actually means that $\mathcal{A}$ goes through the same sequence of states while reading w_i as it did while reading w. But then w_n is the desired word from the statement of the lemma.

To prove this we first assume that $i = 1$. We set $\tau_0' = \tau_0$ and select $d \in S$ such that $\tau_0, d \models \varphi_1$, where φ_1 appears in transition witnessing that $(q_0, \tau_0) \vdash_{a_1, d_1} (q_1, \tau_1)$. Note that this is possible since we have $k + 1$ values at disposal and test only for equality or inequality with a fixed set of k elements. Furthermore, we require that τ_1 and τ_1' are of the same equality type, where τ_1' is obtained from τ_0' as dictated by the transition witnessing that $(q_0, \tau_0) \vdash_{a_1, d_1} (q_1, \tau_1)$ (namely, it is equal to τ_0', or they differ in one position). Again, this is possible since the original

d_1 (from w) could have either been different from all data values in τ_0 or equal to some of them, a choice we can simulate with elements from S. We now set $w_1 = \binom{a_1}{d}$.

Assume now that the claim holds for $i < n$. We prove the claim for $i + 1$. By the induction hypothesis we know that there exists a data word $w_i = \binom{a_1}{d_1^i} \dots \binom{a_i}{d_i^i}$ with data values from S and a sequence of assignments each one obtained from the previous by the condition dictated by the original accepting run that allow $\mathcal{A}$ to go through the states $q_0, q_1, \dots, q_i$. We now pick $d \in S$ such that $\tau_i', d \models \varphi_{i+1}$ and τ_{i+1}', obtained from τ_i' as dictated by the transition in the original run for w, has the same equality type as τ_{i+1}. Note that this is possible since τ_i and τ_i' have the same equality type by the induction hypothesis and we have enough data values at our disposal (again, we have to pick d so that it is in the same relation to data values from τ_i' as d_{i+1} from w was to data values from τ_i, but this is possible since each assignment can remember at most k data values). Now we simply define $w_{i+1} = w_i \cdot \binom{a_{i+1}}{d}$. Note that this w_{i+1} has all the desired properties and can take $\mathcal{A}$ from q_0 to q_{i+1}.

This concludes the proof of the lemma.

4. Regular expressions with memory

In this section we define our first class of regular expressions for data words, called regular expression with memory (REM), and we show that they are equivalent to RA in terms of expressive power. The idea behind them follows closely the equivalence between the standard regular expression and finite state automata. Notice that RA can be pictured as finite state automata whose transitions between states have labels of the form $a[\varphi]{\downarrow}x$ or $a[\varphi]$. Likewise, the building blocks for REM are expressions of the form $a[\varphi]{\downarrow}x$ and $a[\varphi]$.

Definition 4.1 (Regular expressions with memory (REM))**.** *Let Σ be a finite alphabet and $x_1, \dots, x_k$ be variables. Regular expressions with memory (REM) over $\Sigma[x_1, \dots, x_k]$ are defined inductively as follows:*

- *ε and $\emptyset$ are REM;*
- *$a[\varphi]\downarrow x_i$ and $a[\varphi]$ are REM, where $a \in \Sigma$, φ is a condition in $\mathcal{C}_k$, and $x_i \in \{x_1, \dots, x_k\}$;*
- *If e, e_1, e_2 are REM, then so are $e_1 + e_2$, $e_1 \cdot e_2$, and e^*.*

For convenience, when $\varphi = \mathtt{tt}$, we will write a and $a{\downarrow}x$, instead of $a[\mathtt{tt}]$ and $a[\mathtt{tt}]{\downarrow}x$.

Semantics. To define the language expressed by an REM e, we need the following notation. Let e be an REM over $\Sigma[x_1, \ldots x_k]$ and $\nu, \nu' \in \mathcal{F}(x_1, \ldots, x_k)$. Let w be a data word. We define a relation $(e, w, \nu) \vdash \nu'$ inductively as follows.

- $(\varepsilon, w, \nu) \vdash \nu'$ if and only if $w = \varepsilon$ and $\nu' = \nu$.
- $(a[\varphi]{\downarrow}x_i, w, \nu) \vdash \nu'$ if and only if $w = \binom{a}{d}$ and $\nu, d \models \varphi$ and $\nu' = \nu[x_i \leftarrow d]$.
- $(a[\varphi], w, \nu) \vdash \nu'$ if and only if $w = \binom{a}{d}$ and $\nu, d \models \varphi$ and $\nu' = \nu$.
- $(e_1 \cdot e_2, w, \nu) \vdash \nu'$ if and only if there exist w_1, w_2 and ν'' such that $w = w_1 \cdot w_2$ and $(e_1, w_1, \nu) \vdash \nu''$ and $(e_2, w_2, \nu'') \vdash \nu'$.
- $(e_1 + e_2, w, \nu) \vdash \nu'$ if and only if $(e_1, w, \nu) \vdash \nu'$ or $(e_2, w, \nu) \vdash \nu'$.
- $(e^*, w, \nu) \vdash \nu'$ if and only if
  1. $w = \varepsilon$ and $\nu = \nu'$, or
  2. there exist w_1, w_2 and ν'' such that $w = w_1 \cdot w_2$ and $(e, w_1, \nu) \vdash \nu''$ and $(e^*, w_2, \nu'') \vdash \nu'$.

We say that (e, w, ν) infers ν', if $(e, w, \nu) \vdash \nu'$. If $(e, w, \emptyset) \vdash \nu$, then we say that e *induces* ν on data word w. We define $L(e)$ as follows.

$$L(e) \;=\; \{w \mid e \text{ induces } \nu \text{ on } w \text{ for some } \nu\}$$

Example 4.2. *The following two REMs e_1 and e_2:*

$$\begin{aligned} e_1 &= (a{\downarrow}x) \cdot (a[x^{\neq}])^* \\ e_2 &= a^* \cdot (a{\downarrow}x) \cdot a^* \cdot (a[x^{=}]) \cdot a^* \end{aligned}$$

captures precisely the languages $L(\mathcal{A}_1)$ and $L(\mathcal{A}_2)$ in Example 2.2, respectively.

Example 4.3. *Let $\Sigma = \{a, b_1, b_2, \ldots, b_l\}$. Consider the following REM e over $\Sigma[x, y]$:*

$$e \;=\; \Sigma^* \cdot (a{\downarrow}x) \cdot \Sigma^* \cdot (a{\downarrow}y) \cdot \Sigma^* \cdot (\Sigma[x^{=}] + \Sigma[y^{=}]) \cdot (\Sigma[x^{=}] + \Sigma[y^{=}])$$

where $\Sigma[x^{=}]$ stands for $(a[x^{=}] + b_1[x^{=}] + \cdots + b_l[x^{=}])$. The language $L(e)$ consists of data words in which the last two data values occur elsewhere in the word with label a.

The following theorem states that REM and RA are equivalent in expressive power. The proof can be found in Section 4.1.

Theorem 4.4. *REM and RA have the same expressive power in the following sense.*

- *For every REM e over $\Sigma[x_1, \ldots, x_k]$, there exists a k-register RA $\mathcal{A}_e$ such that $L(e) = L(\mathcal{A}_e)$. Moreover, the RA $\mathcal{A}_e$ can be constructed in polynomial time.*
- *For every k-register RA $\mathcal{A}$, there exists an REM $e_{\mathcal{A}}$ over $\Sigma[x_1, \ldots, x_k]$ such that $L(e_{\mathcal{A}}) = L(\mathcal{A})$. Moreover, the REM $e_{\mathcal{A}}$ can be constructed in exponential time.*

Applying Theorem 4.4 and Fact 3.4, we immediately obtain that languages defined by REM are closed under union, intersection, concatenation and Kleene star, but *not* under complement. We also note that Theorems 4.6 and 3.3 imply that the universality and containment problems for REM are undecidable.

The next theorem states that the nonemptiness and the membership problems for REM are PSPACE-complete and NP-complete, respectively. The proofs can be found in Sections 4.2 and 4.3. Note that the hardness results do not follow from Theorems 3.3 and 4.6, since the conversion from RA to REM in Theorem 4.6 takes exponential time.

Theorem 4.5.

- *The nonemptiness problem for REM is* PSPACE*-complete.*
- *The membership problem for REM is* NP*-complete.*

4.1. Proof of Theorem 4.4

In what follows we will need the following notation. For an REM e over $\Sigma[x_1, \ldots, x_k]$ and $\nu, \nu' \in \mathcal{F}(x_1, \ldots, x_k)$, let $L(e, \nu, \nu')$ be the set of all data words w such that $(e, w, \nu) \vdash \nu'$. For an RA $\mathcal{A}$ with k registers, let $L(\mathcal{A}, \nu, \nu')$ be the set of all data words w such that there is an accepting run $(q_0, \nu_0), \ldots, (q_n, \nu_n)$ of $\mathcal{A}$ on w and $\nu_0 = \nu$ and $\nu_n = \nu'$.

We are going to prove the following lemma which immediately implies Theorem 4.4.

Lemma 4.6.

1. *For every REM e over $\Sigma[x_1, \ldots, x_k]$, there exists a k-register RA $\mathcal{A}_e$ such that $L(e, \nu, \nu') = L(\mathcal{A}_e, \nu, \nu')$ for every $\nu, \nu' \in \mathcal{F}(x_1, \ldots, x_k)$. Moreover, the RA $\mathcal{A}_e$ can be constructed in polynomial time.*

2. *For every k-register RA $\mathcal{A}$, there exists an REM $e_{\mathcal{A}}$ over $\Sigma[x_1, \ldots, x_k]$ such that $L(e_{\mathcal{A}}, \nu, \nu') = L(\mathcal{A}, \nu, \nu')$ for every $\nu, \nu' \in \mathcal{F}(x_1, \ldots, x_k)$. Moreover, the REM $e_{\mathcal{A}}$ can be constructed in exponential time.*

The rest of this subsection is devoted to the proof of Lemma 4.6. The structure of the proof follows the standard NFA-regular expressions equivalence, cf. [25], with all the necessary adjustments to handle transitions induced by $a[\varphi]\downarrow x$ or $a[\varphi]$.

We prove the first item by induction on the structure of e.

As before, if $(e, w, \sigma) \vdash \sigma'$, we will write $w \in L(e, \sigma, \sigma')$ and similarly if $\mathcal{A}_e = (Q, q_0, F, \delta)$ started with σ in the registers accepts w with σ' in the registers, we write $w \in L(\mathcal{A}_e, \sigma, \sigma')$.

- If $e = \emptyset$, then $\mathcal{A}_e = (Q, q_0, F, T)$, where $Q = \{q_0\}$ is the set of states, q_0 is the initial state, $F = \emptyset$ is the set of final states and $T = \emptyset$.
- If $e = \varepsilon$, then $\mathcal{A}_e = (Q, q_0, F, T)$, where $Q = \{q_0\}$ is the set of states, q_0 is the initial state, $F = \{q_0\}$ the set of final states and $T = \emptyset$.
- If $e = a[\varphi] \downarrow x_i$, then $\mathcal{A}_e = (Q, q_0, F, , T)$, where $Q = \{q_0, q_1\}$ is the set of states, q_0 is the initial state, $F = \{q_1\}$ the set of final states and $T = \{q_0, a[\varphi]\downarrow x_i \to q_1\}$.
- If $e = a[\varphi]$, then $\mathcal{A}_e = (Q, q_0, F, , T)$, where $Q = \{q_0, q_1\}$ is the set of states, q_0 is the initial state, $F = \{q_1\}$ the set of final states and $T = \{q_0, a[\varphi] \to q_1\}$.
- If $e = e_1 + e_2$ then by the inductive hypothesis we already have automata $\mathcal{A}_{e_1} = (Q_1, s_1, F_1, T_1)$ and $\mathcal{A}_{e_2} = (Q_2, s_2, F_2, T_2)$ with the desired property. The registers of $\mathcal{A}_e$ will be the union of registers of $\mathcal{A}_{e_1}$ and $\mathcal{A}_{e_2}$. To obtain the desired automaton we set $\mathcal{A}_e = (Q, q_0, F, T)$, where:
  - $Q = Q_1 \cup Q_2 \cup \{q_0\}$, where q_0 is a new state,
  - $F = F_1 \cup F_2$,
  - To T we add all transitions from $\mathcal{A}_{e_1}$ and $\mathcal{A}_{e_2}$ and in addition, for every transition $q, a[\varphi]\downarrow x_i \to q' \in T_1 \cup T_2$, where $q = s_1$, or $q = s_2$, we add a transition $q_0, a[\varphi]\downarrow x_i \to q'$ and similarly for transitions of the form $q, a[\varphi] \to q'$.

- If $e = e_1 \cdot e_2$ then by the inductive hypothesis we already have automata $\mathcal{A}_{e_1} = (Q_1, s_1, F_1, T_1)$ and $\mathcal{A}_{e_2} = (Q_2, s_2, F_2, T_2)$ with the desired property. The registers of $\mathcal{A}_e$ will be the union of registers of $\mathcal{A}_{e_1}$ and $\mathcal{A}_{e_2}$. To obtain the desired automaton $\mathcal{A}_e = (Q, q_0, F, T)$ we distinguish two cases:

  1. If $s_1 \notin F_1$ we set
     - $Q = Q_1 \cup Q_2$,
     - $F = F_2$,
     - $q_0 = s_1$
     - To T we add all transitions from $\mathcal{A}_{e_1}$ and $\mathcal{A}_{e_2}$ and in addition, for every transition $q, a[\varphi]\downarrow x_i \to q' \in T_1$, where $q' \in F_1$, we add a transition $q, a[\varphi]\downarrow x_i \to s_2$ and similarly for $q, a[\varphi] \to q'$ with $q' \in F_1$.
  2. If $s_1 \in F_1$ we set
     - $Q = Q_1 \cup Q_2$,
     - $F = \begin{cases} F_2 & \text{if } s_2 \notin F_2 \\ F_1 \cup F_2 & \text{if } s_2 \in F_2 \end{cases}$,
     - $q_0 = s_1$
     - To T we add all transitions from $\mathcal{A}_{e_1}$ and $\mathcal{A}_{e_2}$ and in addition, for every transition $s_2, a[\varphi]\downarrow x_i \to q' \in T_2$, we add a transition $q, a[\varphi]\downarrow x_i \to q'$, for each $q \in F_1$ and analogously for $s_2, a[\varphi] \to q'$.

- If $e = e_1^*$ then by the inductive hypothesis we already have the automaton $\mathcal{A}_{e_1} = (Q_1, s_1, F_1, T_1)$ with the desired property. The registers of $\mathcal{A}_e$ will be equal to the registers of $\mathcal{A}_{e_1}$. To obtain the desired automaton we set $\mathcal{A}_e = (Q, q_0, F, T)$, where:
  - $Q = Q_1 \cup \{q_0\}$, where q_0 is a new state,
  - $F = F_1 \cup \{q_0\}$,
  - To T we add all transitions from $\mathcal{A}_{e_1}$ and in addition, for every transition $s_1, a[\varphi]\downarrow x_i \to q' \in T_1$, we add a transition $q_0, a[\varphi]\downarrow x_i \to q'$ to T. Now for every transition $q, a[\varphi]\downarrow x_i \to q' \in T$ (note that we now have transitions from q_0 as well), where $q' \in F_1$, we add $q, a[\varphi]\downarrow x_i \to q_0$ to T. Transitions of the form $q, a[\varphi] \to q'$ are treated analogously.

In all cases it is straightforward to check that the constructed automaton has the desired property. The polynomial time bound follows immediately from the construction.

Next we move onto the second claim of the theorem.

To prove this we will have to introduce generalized register automata (GRA for short) over data words. The difference from usual register automata will be that we allow arrows to be labelled by arbitrary regular expressions with memory. I.e. our arrows are now not labelled only by $a[\varphi]\downarrow x_i$, or $a[\varphi]$, but by any regular expression with memory. The transition relation is called δ and is defined as $\delta \subseteq Q \times REM(\Sigma[x_1, \ldots, x_k]) \times Q$, where $REM(\Sigma[x_1, \ldots, x_k])$ denotes the set of all regular expressions with memory over $\Sigma[x_1, \ldots, x_k]$. In addition to that we also specify that we have a single initial state with no incoming arrows and a single final state with no outgoing arrows. Note that we also allow ε-transitions.

The only difference is how we define acceptance.

A GRA $\mathcal{A}$ accepts data word w if $w = w_1 \cdot w_2 \cdot \ldots \cdot w_k$(where each w_i is a data word) and there exists a sequence $c_0 = (q_0, \tau_0), \ldots, c_k = (q_k, \tau_k)$ of configurations of $\mathcal{A}$ on w such that:

1. q_0 is the initial state,
2. q_k is a final state,
3. for each i we have $(e_i, w_i, \tau_i) \vdash \tau_{i+1}$(i.e. $w_i \in L(e_i, \tau_i, \tau_{i+1})$), for some e_i such that (q_i, e_i, q_{i+1}) is in the transition relation for $\mathcal{A}$.

We can now prove the equivalence of register automata and regular expressions with memory by mimicking the construction used to prove equivalence between ordinary finite state automata and regular expressions (over strings). Since we use the same construction we will get an exponential blow-up, just like for finite state automata.

Just as in the finite state case we first convert $\mathcal{A}$ into a GRA by adding a new initial state (connected to the old initial state by an ε-arrow) and a new final state (connected to the old end states by incoming ε-arrows). We also assume that this automaton has only a single arrow between every two states (we achieve this by replacing multiple arrows by union of expressions). It is clear that this GRA recognizes the same language of data words as $\mathcal{A}$.

Next we show how to convert this automaton into an equivalent expression. We will use the following recursive procedure which rips out one state at a time from the automaton and stops when we end with only two states (note that this procedure is taken from [25]).

CONVERT(G)

1. Let n be the number of states of G.

2. If $n = 2$ then G contains only a start state and an end state with a single arrow connecting them. This arrow has an expression R written on it. Return R.

3. If $n > 2$ select any state q_{rip}, different from q_{start} and q_{end} and modify G in the following manner to obtain G' with one less state. The new set of states is $Q' = Q - \{q_{rip}\}$ and for any $q_i \in Q' - \{q_{accept}\}$ and any $q_j \in Q' - \{q_{start}\}$ we define $\delta'(q_i, q_j) = (R_1)(R_2)^*(R_3) + R_4$, where $R_1 = \delta(q_i, q_{rip}), R_2 = \delta(q_{rip}, q_{rip}), R_3 = \delta(q_{rip}, q_j)$ and $R_4 = \delta(q_i, q_j)$. The initial and final state remain the same.

4. Return CONVERT(G').

We now prove that for any GRA G the expression CONVERT(G) and GRA G recognize the same language of data words. We do so by induction on the number n of states of our GRA G. If $n = 2$ then G has only a single arrow from initial to final state and by definition of acceptance for GRA the expression on this arrow recognizes the same language as G.

Assume now that the claim is true for all automatons with $n - 1$ states. Let G be an automaton with n states. We prove that G is equivalent to automaton G' obtained in the step 3 of our CONVERT algorithm. Note that this completes the induction.

To see this assume first that $w \in L(G, \sigma, \sigma')$, i.e. G with initial assignment σ has an accepting run on w ending with σ' in the registers. This means that there exists a sequence of configurations $c_0 = (q_0, \tau_0), \ldots, c_k = (q_k, \tau_k)$ such that $w = w_1 w_2 \ldots w_k$, where each w_i is a data word (with possibly more than one symbol), $\tau_0 = \sigma, \tau_k = \sigma'$ and $(\delta(q_{i-1}, q_i), w_i, \tau_{i-1}) \vdash \tau_i$, for $i = 1, \ldots, k$. (Here we use the assumption that we only have a single arrow between any two states).

If none of the states in this run are q_{rip}, then it is also an accepting run in G', so $w \in L(G, \sigma, \sigma')$, since all the arrows present here are also in G'.

If q_{rip} does appear we have the following in our run

$$c_i = (q_i, \tau_i), c_{rip} = (q_{rip}, \tau_{i+1}), \ldots, c_{rip} = (q_{rip}, \tau_{j-1}), c_j = (q_j, \tau_j).$$

If we show how to unfold this to a run in G' we are done (if this appears more than once we apply the same procedure).

Since this is the case we know (by the definition of accepting run) that $(R_1, w_{i+1}, \tau_i) \vdash \tau_{i+1}, (R_2, w_{i+2}, \tau_{i+1}) \vdash \tau_{i+2}, (R_2, w_{i+3}, \tau_{i+2}) \vdash \tau_{i+3}, \ldots, (R_2, w_{j-1}, \tau_{j-2}) \vdash \tau_{j-1}$ and $(R_3, w_j, \tau_{j-1}) \vdash \tau_j$, where $R_1 = \delta(q_i, q_{rip}), R_2 = \delta(q_{rip}, q_{rip}), R_3 = \delta(q_{rip}, q_j)$. Note that this simply means that $((R_1)(R_2)^*(R_3), w_i w_{i+1} \ldots w_j, \sigma) \vdash \sigma'$, so G' can jump from c_i to c_j using only one transition.

Conversely, suppose that $w \in L(G', \sigma, \sigma')$. This means that there is a computation of G' starting with σ and ending with σ' as register assignments. We know that each arrow in G' from q_i to q_j goes either directly (in which case it is already in G) or through q_{rip} (in which case we use the definition of acceptance by regular expressions to unravel this word into part recognized by G). In either case we get an accepting run of G on w.

To see that this gives the desired result observe that we can always convert register automaton into an equivalent GRA and use CONVERT to obtain a regular expression with memory recognizing the same language.

4.2. The PSPACE*-completeness of the nonemptiness problem for REM*

By Theorem 4.6, we can convert REM to RA in polynomial time. By Theorem 3.3, the nonemptiness problem for RA is PSPACE. Hence, the PSPACE upper bound follows.

In the remaining of the proof, we are going to establish the PSPACE-hardness.

The PSPACE-hardness is via a reduction from the nonuniversality problem of finite state automata. The nonuniversality problem asks, given a finite state automaton $\mathcal{A}$ as input, to decide whether $L(\mathcal{A}) \neq \Sigma^*$. Assume we are given a regular automaton $\mathcal{A} = (Q, \Sigma, \delta, q_1, F)$, where $Q = \{q_1, \ldots, q_n\}$ and $F = \{q_{i_1}, \ldots, q_{i_k}\}$.

Since we are trying to demonstrate nonuniversality of the automaton $\mathcal{A}$ we simulate reachability checking in the powerset automaton for $\overline{\mathcal{A}}$. To do so we designate two distinct data values, t and f, and code each state of the powerset automaton as an n-bit sequence of t/f values, where the ith bit of the sequence is set to t if the state q_i is included in our state of $\overline{\mathcal{A}}$. Since we are checking reachability we will need only to remember the current and the next state of $\overline{\mathcal{A}}$. In what

follows we will code those two states using variables $s_1, \ldots, s_n$ and $t_1, \ldots, t_n$ and refer to them as the current state tape and the next state tape. Our expression e will code data words that describe successful runs of $\overline{\mathcal{A}}$ by demonstrating how one can move from one state of this automaton to another (as witnessed by their codes in current state tape and next state tape), starting with the initial and ending in a final state.

We will define several expressions and explain their role. We will use two sets of variables, s_1 through s_n and $t_1, \ldots, t_n$ to denote the current state tape and the next state tape. All of these variables will only contain two values, t and f, which are bound in the beginning.

The first expression we need is:

$$\texttt{init} := (a{\downarrow}t) \cdot (a[t^{\neq}]{\downarrow}f) \cdot (a[t^{=}]{\downarrow}s_1) \cdot (a[f^{=}]{\downarrow}s_2) \ldots (a[f^{=}]{\downarrow}s_n).$$

This expression codes two different values as t and f and initializes current state tape to contain encoding of initial state (the one where only the initial state from $\mathcal{A}$ can be reached). That is, a data word is in the language of this expression if and only if it starts with two different data values and continues with n data values that form a sequence in 10^*, where 1 represents the value assigned to t and 0 the one assigned to f.

The next expression we use is as follows:

$$\texttt{end} := a[f^{=} \wedge s_{i_1}^{=}] \cdot a[f^{=} \wedge s_{i_2}^{=}] \cdot \cdots a[f^{=} \wedge s_{i_k}^{=}], \text{ where } F = \{q_{i_1}, \ldots, q_{i_k}\}.$$

This expression is used to check that we have reached a state not containing any final state from the original automaton. That is, a data word is in $L(\texttt{end})$ if and only if it consists of k data values, all equal to f and where value stored in s_{i_j} also equals f, for $j = 1 \ldots k$.

Next we define expressions that will reflect updating of the next state tape according to the transition function of $\mathcal{A}$. Assume that $\delta(q_i, b) = \{q_{j_1}, \ldots, q_{j_l}\}$. We define

$$u_{\delta(q_i,b)} := \big((a[t^{=} \wedge s_i^{=}]) \cdot (a[t^{=}]{\downarrow}t_{j_1}) \ldots (a[t^{=}]{\downarrow}t_{j_l})\big) + a[f^{=} \wedge s_i^{=}].$$

Also, if $\delta(q_i, b) = \emptyset$ we simply put $u_{\delta(q_i,b)} := \varepsilon$.

This expression will be used to update the next state tape by writing true to corresponding variables if the state q_i is tagged with t on the current state tape (and thus contained in the current state of $\overline{\mathcal{A}}$). If it is false we skip the update.

Since we have to define update according to all transitions from all the states corresponding to chosen letter we get:

$$\texttt{update} := \bigvee_{b \in \Sigma} \bigwedge_{q_i \in Q} u_{\delta(q_i,b)}.$$

This simply states that we non deterministically pick the next symbol of the word we are guessing and move to the next state accordingly.

We still have to ensure that the tapes are copied at the beginning and end of each step, so we define:

$$\texttt{step} := \big((a[f^{=}]{\downarrow}t_1) \dots (a[f^{=}]{\downarrow}t_n)\big) \cdot \texttt{update} \cdot \big((a[t_1^{=}]{\downarrow}s_1) \dots (a[t_n^{=}]{\downarrow}s_n)\big).$$

This simply initializes the next state tape at the beginning of each step, proceeds with the update and copies the next state tape to the current state tape.

Finally we have

$$e := \texttt{init} \cdot (\texttt{step})^* \cdot \texttt{end}.$$

We claim that for $L(e) \neq \emptyset$ if and only if $L(\mathcal{A}) \neq \Sigma^*$.

Assume first that $L(\mathcal{A}) \neq \Sigma^*$. This means that there is a path from the initial to the final state in the powerset automaton for $\overline{\mathcal{A}}$. That is, there is a word w from Σ^* not in the language of $\mathcal{A}$. This path can in turn be described by pairs of assignment of values t/f to the current state tape and the next state tape, where each transition is witnessed by the corresponding letter of the alphabet. But then the word that belongs to $L(e)$ is the one that first initializes the stable tape (i.e. the variables $s_1, \dots, s_n$) to initial state of the powerset automaton, then runs the updates of the tape according to w and finally ends in a state where all variable corresponding to end states of $\mathcal{A}$ are tagged f.

Conversely, each word in $L(e)$ corresponds to a run of the powerset automaton for $\overline{\mathcal{A}}$. That is, the part of word corresponding to `init` sets the initial state. Then the part of this word that corresponds to $\texttt{step}^*$ corresponds to updating our tapes in a way that properly codes one step of the powerset automaton. Finally, `end` denotes that we have reached a state where all end states of $\mathcal{A}$ have been tagged by f, and therefore an accepting state for $\overline{\mathcal{A}}$.

4.3. Proof of the NP*-completeness of the membership problem for REM*

By Theorem 4.6, an REM e can be translated to an RA $\mathcal{A}_e$ in polynomial time. Since the membership problem for RA is in NP by Theorem 3.3, the NP upper bound follows. For the NP-hardness, we do a reduction from 3-SAT.

Let $\varphi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \ldots \wedge (a_k \vee b_k \vee c_k)$, be an arbitrary 3-CNF formula. We will construct a data word w and a regular expression with memory e, both of length linear in the length of φ, such that φ is satisfiable if and only if $w \in L(e)$.

Let $x_1, x_2, \ldots, x_n$ be all the variables occurring in φ. We define w as the following data word:

$$w = \left(\binom{a}{0}\binom{b}{1}\right)^n \left(\binom{a_1}{d_{a_1}}\binom{b_1}{d_{b_1}}\binom{c_1}{d_{c_1}}\right) \cdots \left(\binom{a_k}{d_{a_k}}\binom{b_k}{d_{b_k}}\binom{c_k}{d_{c_k}}\right),$$

where $d_{a_i} = 1$, if $a_i = x_j$, for some $j \in \{1, \ldots n\}$ and 0, if $a_i = \overline{x_j}$ and similarly for d_{b_i}, d_{c_i} (note that every a_i, b_i, c_i is of the form x_j, or $\overline{x_j}$, so this is well defined).

Also note that we are using a_i, b_i, c_i both for literals in φ and for letters of our finite alphabet, but this should not cause any confusion. The idea behind this data word is that with the first part that corresponds to the variables, i.e. with $(\binom{a}{0}\binom{b}{1})^n$, we guess a satisfying assignment and the next part corresponds to each conjunct in φ and its data value is set such that if we stop at any point for comparison we get a true literal in this conjunct.

We now define e as the following regular expression with memory:

$$e = (a{\downarrow}x_1 + ab{\downarrow}x_1) \cdot b^* \cdot (a{\downarrow}x_2 + ab{\downarrow}x_2) \cdot b^* \cdot (a{\downarrow}x_3 + ab{\downarrow}x_3) \cdots \\ b^* \cdot (a{\downarrow}x_n + ab{\downarrow}x_n) \cdot b^* \cdot \text{clause}_1 \cdot \text{clause}_2 \ldots \text{clause}_k,$$

where each clause_i corresponds to the i-th conjunct of φ in the following manner.

If ith conjunct uses variables $x_{j_1}, x_{j_2}, x_{j_3}$(possibly with repetitions), then

$$\text{clause}_i = a_i[x_{j_1}^{=}] \cdot b_i \cdot c_i + a_i \cdot b_i[x_{j_2}^{=}] \cdot c_i + a_i \cdot b_i \cdot c_i[x_{j_3}^{=}].$$

We now prove that φ is satisfiable if and only if $w \in L(e)$.

Assume first that φ is satisfiable. Then there is a way to assign a value to each x_i such that for every conjunct in φ at least one literal is true. This means that we can traverse the first part of w to chose the corresponding values for variables bounded in e. Now with this choice we can make one of the literals in each conjunct true, so we can traverse every clause_i using one of the tree possibilities.

Assume now that $w \in L(e)$. This means that after choosing the data values for variables (and thus a valuation for φ, since all data values are either 0 or 1), we are able to traverse the second part of w using these values. This means that for every clause_i there is a letter after which the data value is the same as the one bounded to the corresponding variable. Since data values in the second part of w imply that a literal in the corresponding conjunct of φ evaluates to 1, we know that this valuation satisfies our formula φ.

5. Regular expressions with binding

In this section we define and study a class of expressions whose variables adhere to the usual scoping rules familiar from programming languages or first order logic.

The idea is again to use variables that store data values and then compare them using conditions. The storing of a value, however, will bind it only to the scope of the variable used, unlike in regular expressions with memory.

Conditions are defined in the same manner as in Section 2. Next we define regular expressions with binding.

As mentioned in the introduction, regular expressions with memory have a similar syntax but rather different semantics than REWBs. They are built using $a \downarrow x$, concatenation, union and Kleene star (see Section 4). That is, no binding is introduced with $a \downarrow x$; rather it directly matches the operation of putting a value in a register. In contrast, REWBs use proper bindings of variables; expression $a \downarrow_x$ appears only in the context $a \downarrow_x .\{r\}$ where it binds x inside the expression r only. Theorem 4.6 states that expressions with memory and register automata are one and the same in terms of expressive power. Here we show that REWBs, on the other hand, are strictly weaker. Therefore, proper binding of variables comes with a cost – albeit small – in terms of expressiveness.

Definition 5.1. *Let Σ be a finite alphabet and $\{x_1, \ldots, x_k\}$ a finite set of variables.* Regular expressions with binding *(REWB) over $\Sigma[x_1, \ldots, x_k]$ are defined inductively as follows:*

$$r \;:=\; \varepsilon \;\mid\; a \;\mid\; a[\varphi] \;\mid\; r + r \;\mid\; r \cdot r \;\mid\; r^* \;\mid\; a \downarrow_{x_i} .\{r\}$$

where $a \in \Sigma$ and φ is a condition in $\mathcal{C}_k$.

A variable x_i is bound if it occurs in the scope of some $\downarrow_{x_i}$ operator and free otherwise. More precisely, free variables of an expression are defined inductively: ε and a have no free variables, in $a[\varphi]$ all variables occurring in φ are free, in $r_1 + r_2$ and $r_1 \cdot r_2$ the free variables are those of r_1 and r_2, the free variables of r^* are those of r, and the free variables of $a \downarrow_{x_i} .\{r\}$ are those of r except x_i. We will write $r(x_1, \ldots, x_l)$ if $x_1, \ldots, x_l$ are the free variables in r.

Semantics. Let $r(\overline{x})$ be an REWB over $\Sigma[x_1, \ldots, x_k]$. A valuation $\nu \in \mathcal{F}(x_1, \ldots, x_k)$ is compatible with r, if $\nu(\overline{x})$ is defined.

A regular expression $r(\overline{x})$ over $\Sigma[x_1, \ldots, x_k]$ and a valuation $\nu \in \mathcal{F}(x_1, \ldots, x_k)$ compatible with r define a language $L(r, \nu)$ of data words as follows.

- If $r = a$ and $a \in \Sigma$, then $L(r, \nu) = \{\binom{a}{d} \mid d \in \mathcal{D}\}$.
- If $r = a[\varphi]$, then $L(r, \nu) = \{\binom{a}{d} \mid d, \nu \models \varphi\}$.
- If $r = r_1 + r_2$, then $L(r, \nu) = L(r_1, \nu) \cup L(r_2, \nu)$.
- If $r = r_1 \cdot r_2$, then $L(r, \nu) = L(r_1, \nu) \cdot L(r_2, \nu)$.
- If $r = r_1^*$, then $L(r, \nu) = L(r_1, \nu)^*$.
- If $r = a \downarrow_{x_i} .\{r_1\}$, then $L(r, \nu) = \bigcup_{d \in \mathcal{D}} \left\{\binom{a}{d}\right\} \cdot L(r_1, \nu[x_i \leftarrow d])$.

A REWB r defines a language of data words as follows.

$$L(r) \quad = \bigcup_{\nu \text{ compatible with } r} L(r, \nu).$$

In particular, if r is without free variables, then $L(r) = L(r, \emptyset)$. We will call such REWBs *closed*.

Example 5.2. *The following two examples are the REWB equivalent of the languages in Example 2.2.*

- *The language L_1 that consists of data words where the data value in the first position is different from the others is given by: $a \downarrow_x .\{(a[x^{\neq}])^*\}$.*
- *The language L_2 that consists of data words where there are two positions with the same data value is define by: $a^* \cdot a \downarrow_x .\{a^* \cdot a[x^{=}]\} \cdot a^*$.*

A straightforward induction on expressions shows that REWB languages are contained in RA, hence, in REM. It also follows from the definition that REWB languages are closed under union, concatenation and Kleene star. However, similar to RA languages, they are not closed under complement. Consider the language L_2 in Example 5.2. The complement of this language, where all data values are different, is well known not to be definable by register automata [1].

The theorem below states that REWB languages are not closed under intersection. The proof can be found in Section 5.1.

Theorem 5.3. *The REWB languages are not closed under intersection.*

The separating example also shows that REWB is strictly weaker than REM.

Corollary 5.4. *In terms of expressive power REWB is strictly weaker than RA.*

Proof. Consider the language $L_1 \cap L_2$ defined in Section 5.1. It is easy to see that this language can be defined using RA. □

We note that the separating example is rather intricate, and certainly not a natural language one would think of. In fact, all natural languages definable with register automata that we used here as examples – and many more, especially those suitable for graph querying – are definable by REWBs.

The next theorem states the complexity behaviour of REWB.

Theorem 5.5.

- *The nonemptiness problem for REWB is* NP*-complete.*
- *The universality problem for REWB is undecidable.*

Recall the nonemptiness problem for both RA and REM is PSPACE-complete. Introducing the proper binding, the complexity of the nonemptiness problem for REWB drops to NP-complete. The proof of NP-completeness can be found in Section 5.2 and the undecidability is established in Section 5.3.

5.1. Proof of Theorem 5.3

Let L_1 the language consists of data words of the form:

$$\binom{a}{d_1}\binom{a}{d_2}\binom{a}{d_3}\binom{a}{d_4}\binom{a}{d_5}\binom{a}{d_6}\binom{a}{d_7}\binom{a}{d_8}\cdots\cdots\cdots\binom{a}{d_{4n}}$$

where $d_2 = d_5, d_6 = d_9, \ldots, d_{4n-6} = d_{4n-3}$.

Let L_2 be the language as above, but $d_4 = d_7, d_8 = d_{11}, \ldots, d_{4n-4} = d_{4n-1}$.

In particular, $L_1 \cap L_2$ is the language consists of data words of the form:

$$\binom{a}{d_1}\binom{a}{d_2}\binom{a}{e_1}\binom{a}{e_2}\binom{a}{d_2}\binom{a}{d_3}\binom{a}{e_2}\binom{a}{e_3}\cdots\cdots\cdots \binom{a}{d_{m-2}}\binom{a}{d_{m-1}}\binom{a}{e_{m-2}}\binom{a}{e_{m-1}}\binom{a}{d_{m-1}}\binom{a}{d_m}\binom{a}{e_{m-1}}\binom{a}{e_m}$$

Both L_1 and L_2 are REWB languages. In the following we are going to show the following.

Lemma 5.6. *$L_1 \cap L_2$ is not a REWB language.*

Note that for simplicity we prove the theorem for the case of REWBs that use only conditions of the form $\varphi = x_i^{=}$, or $\varphi = x_i^{\neq}$ (that is, we allow only a single

comparison per condition). It is straightforward to see that the same proof works in the case of REWBs that use multiple comparisons in one condition.

The proof is rather technical and will require a few auxiliary notions. Let r be an REWB over $\Sigma[x_1, \ldots, x_k]$. A *derivation tree* t with respect to r is a tree whose internal nodes are labeled with (r', ν) where r' is an subexpression of r and $\nu \in \mathcal{F}(x_1, \ldots, x_k)$ constructed as follows. The root node is labeled with $(e, \emptyset)$. The other nodes are labeled as follows. For a node u labeled with (r', ν), its children are labeled as follows.

- If $r' = a$, then u has only one child: a leaf node labeled with $\binom{a}{d}$ for some $d \in \mathcal{D}$.
- If $r' = a[\varphi]$, then u has only one child: a leaf node labeled with $\binom{a}{d}$ such that $d, \nu \models \varphi$.
- If $r' = r_1 + r_2$, then u has only one child: a leaf node labeled with either (r_1, ν) or (r_2, ν).
- If $r' = r_1 \cdot r_2$, then u has only two children: the left child is labeled with (r_1, ν) and the right child is labeled with (r_2, ν).
- If $r' = r_1^*$, then u has either only one child: a leaf node labeled with ϵ; or at least one child labeled with (r_1, ν).
- If $r' = a \downarrow_x .\{r_1\}$, then u has only two children: the left child is labeled with $\binom{a}{d}$ and the right child is labeled with $(r_1, \nu[x \leftarrow d])$, for some data value $d \in \mathcal{D}$.

A derivation tree t defines a data word $w(t)$ as the word read on the leaf nodes of t from left to right.

Proposition 5.7. *For every REWB r, the following holds. A data word $w \in L(r, \emptyset)$ if and only if there exists a derivation tree t w.r.t. r such that $w = w(t)$.*

Proof. We start with the "only if" direction. Suppose that $w \in L(r, \emptyset)$. By induction on the length of e, we can construct the derivation tree t such that $w = w(t)$. It is a rather straightforward induction, where the induction step is based on the recursive definition of REWB, where r is either a, $a[x^=]$, $a[x^{\neq}]$, $r_1 + r_2$, $r_1 \cdot r_2$, r_1^* or $a \downarrow_x .\{r_1\}$.

Now we prove the "if" direction.

For a node u in a derivation tree t, the word induced by the node u is the subword made up of the leaf nodes in the subtree rooted at u. We denote such subword by $w_u(t)$.

We are going to show that for every node u in t, if u is labeled with (r', ν), then $w_u(t) \in L(r', \nu)$. This can be proved by induction on the *height* of the node u, which is defined as follows.

- The height of a leaf node is 0.
- The height of a node u is the maximum between the heights of its children nodes plus one.

It is a rather straightforward induction, where the base case is the nodes with zero height and the induction step is carried on nodes of height h with the induction hypothesis assumed to hold on nodes of height $< h$. □

Suppose $w(t) = w_1 w_u(t) w_2$, the *index pair* of the node u is the pair of integers (i, j) such that $i = \text{length}(w_1) + 1$ and $j = \text{length}(w_1 w_u(t))$.

A derivation tree t induces a binary relation R_t as follows.

$$R_t = \{(i, j) \mid (i, j) \text{ is the index pair of a node } u \text{ in } t \text{ labeled with } a \downarrow_{x_l} .\{r'\} \quad \}.$$

Note that R_t is a partial function from the set $\{1, \ldots, \text{length}(w(t))\}$ to itself, where if $R_t(i)$ is defined, then $i < R_t(i)$.

For a pair $(i, j) \in R_t$, we say that the variable x is associated with (i, j), if (i, j) is the index pair of a node u in t labeled with a label of the form $a \downarrow_x .\{r'\}$. Two binary tuples (i, j) and (i', j'), where $i < j$ and $i' < j'$, *cross each other* if either $i < i' < j < j'$ or $i' < i < j' < j$.

Proposition 5.8. *For any derivation tree t, the binary relation R_t induced by it does not contain any two pairs (i, j) and (i', j') that cross each other.*

Proof. Suppose $(i, j), (i', j') \in R_t$. Then let u and u' be the nodes whose index pairs are (i, j) and (i', j'), respectively. There are two cases.

- The nodes u and u' are descendants of each other.
  Suppose u is a descendant of u'. Then, we have $i' < i < j < j'$.
- The nodes u and u' are not descendants of each other.
  Suppose the node u' is on the right side of u, that is, $w_{u'}(t)$ is on the right side of $w_u(t)$ in w. Then we have $i < j < i' < j'$.

In either case (i, j) and (i', j') do not cross each other. This completes the proof of our claim. □

Now we are ready to show that $L_1 \cap L_2$ is not defined by any REWB. Suppose to the contrary that there is an REWB r over $\Sigma[x_1, \ldots, x_k]$ such that $L(r) = L_1 \cap L_2$, where $\Sigma = \{a\}$. Consider the following word w, where $m = k + 2$:

$$w := \binom{a}{d_0}\binom{a}{d_1}\binom{a}{e_0}\binom{a}{e_1}\binom{a}{d_1}\binom{a}{d_2}\binom{a}{e_1}\binom{a}{e_2}\cdots\cdots\cdots \binom{a}{d_{m-2}}\binom{a}{d_{m-1}}\binom{a}{e_{m-2}}\binom{a}{e_{m-1}}\binom{a}{d_{m-1}}\binom{a}{d_m}\binom{a}{e_{m-1}}\binom{a}{e_m}$$

where $d_0, d_1, \ldots, d_m, e_0, e_1, \ldots, e_m$ are pairwise different.

Let t be the derivation tree of w. Consider the binary relation R_t and the following sets A and B.

$$\begin{aligned} A &= \{2, 6, 10, \ldots, 4m-6\} \\ B &= \{4, 8, 12, \ldots, 4m-4\} \end{aligned}$$

That is, the set A contains the first positions of the data values $d_1, \ldots, d_{m-1}$s, and the set B the first positions of the data values $e_1, \ldots, e_{m-1}$s.

Claim 5.9. *The relation R_t is a function on $A \cup B$. That is, for every $h \in A \cup B$, there is h' such that $(h, h') \in R_t$.*

Proof. Suppose there exists $h \in A \cup B$ such that $R_t(h)$ is not defined. Assume that $h \in A$ and l be such that $h = 4l - 2$. If $R_t(h)$ is not defined, then for any valuation ν found in the nodes in t, $d_l \notin \text{Image}(\nu)$. So, the word

$$w'' = \binom{a}{d_0}\binom{a}{d_1}\binom{a}{e_0}\binom{a}{e_1}\cdots\cdots\binom{a}{d_{l-1}}\binom{a}{f}\binom{a}{e_{l-1}}\binom{a}{e_l}\binom{a}{d_l}\binom{a}{d_{l+1}}\cdots\cdots$$

is also in $L(r)$, where f is a new data value. That is, the word w'' is obtained by replacing the first appearance of d_l with f. Now $w'' \notin L_1 \cap L_2$, hence, contradicts the fact that $L(r) = L_1 \cap L_2$. The same reasoning goes for the case if $h \in B$. This completes the proof of our claim. □

Remark 1. *Without loss of generality, we can assume that each variable in the REWB r is introduced only once. Otherwise, we can rename the variable.*

Claim 5.10. *There exist $(h_1, h_2), (h'_1, h'_2) \in R_t$ such that $h_1 < h_2 < h'_1 < h'_2$ and $h_1, h'_1 \in A$ and both $(h_1, h_2), (h'_1, h'_2)$ have the same associated variable.*

Proof. The cardinality $|A| = k + 1$. So there exists a variable $x \in \{x_1, \ldots, x_k\}$ and $(h_1, h_2), (h'_1, h'_2) \in R_t$ such that $(h_1, h_2), (h'_1, h'_2)$ are associated with the variable x. By Remark 1, no variable is written twice in e, so the nodes u, u' associated with $(h_1, h_2), (h'_1, h'_2)$ are not descendants of each other, so we have $h_1 < h_2 < h'_1 < h'_2$, or $h'_1 < h'_2 < h_1 < h_2$. This completes the proof of our claim. □

Claim 5.11 below immediately implies that Lemma 5.6.

Claim 5.11. *There exists a word $w'' \notin L_1 \cap L_2$, but $w'' \in L(r)$.*

Proof. The word w'' is constructed from the word w. By Claim 5.10, there exist $(h_1, h_2), (h'_1, h'_2) \in R_t$ such that $h_1 < h_2 < h'_1 < h'_2$ and $h_1, h'_1 \in A$ and both h_1, h'_1 have the same associated variable.

By definition of the language $L_1 \cap L_2$, between h_1 and h'_1, there exists an index $l \in B$ such that $h_1 < l < h'_1$. (Recall that the set A contains the first positions of the data values $d_1, \ldots, d_{m-1}$s, and the set B the first positions of the data values $e_1, \ldots, e_{m-1}$s.)

Let h be the maximum of such indices. The index h is not the index of the last e, hence $R_t(h)$ exists and $R_t(h) < h_2$, by Proposition 5.8. Now the data value in $R_t(h)$ is different from the data value in position h. To get w'', we change the data value in the position h with a new data value f, and it will not change the acceptance of the word w'' by the REWB r.

However, the word w''

$$w'' = \binom{a}{d_0}\binom{a}{d_1}\binom{a}{e_0}\binom{a}{e_1}\cdots\cdots\binom{a}{e_{l-1}}\binom{a}{f}\cdots\binom{a}{e_l}\binom{a}{e_{l+1}}\cdots\cdots$$

is not in $L_1 \cap L_2$, by definition. Thus, this completes the proof of our claim. □

This completes our proof of Lemma 5.6.

Since both L_1 and L_2 are easily definable by a REWB using only one variable, this completes the proof of Theorem 5.3.

5.2. Proof of NP*-completeness for the nonemptiness problem for REWB*

In order to prove the NP-upper bound from the theorem we will first show that if there is a word accepted by a REWB, then there is also a word accepted that is no longer than the REWB itself.

Proposition 5.12. *For every REWB r over $\Sigma[x_1, \ldots, x_k]$ and every valuation ν compatible with r, if $L(r, \nu) \neq \emptyset$, then there exists a data word $w \in L(r, \nu)$ of length $\mathcal{O}(|r|)$.*

Proof. The proof is by induction on the length of r. The basis is when the length of r is 1. There are two cases: $a[\varphi]$ and a; and it is trivial that our proposition holds.

Let r be an REWB and ν a valuation compatible with r. For the induction hypothesis, we assume that our proposition holds for all REWBs of shorter length

than r. For the induction step, we prove our proposition for r. There are four cases.

- Case 1: $r = r_1 + r_2$.
  If $L(r, \nu) \neq \emptyset$, then by the induction hypothesis, either $L(r_1, \nu)$ or $L(r_2, \nu)$ are not empty. So, either

  - there exists $w_1 \in L(r_1, \nu)$ such that $|w_1| = \mathcal{O}(|r_1|)$; or
  - there exists $w_2 \in L(r_2, \nu)$ such that $|w_2| = \mathcal{O}(|r_2|)$.

  Thus, by definition, there exists $w \in L(r, \nu)$ such that $|w| = \mathcal{O}(|r|)$.
- Case 2: $r = r_1 \cdot r_2$.
  If $L(r, \nu) \neq \emptyset$, then by the definition, $L(r_1, \nu)$ and $L(r_2, \nu)$ are not empty. So by the induction hypothesis

  - there exists $w_1 \in L(r_1, \nu)$ such that $|w_1| = \mathcal{O}(|r_1|)$; and
  - there exists $w_2 \in L(r_2, \nu)$ such that $|w_2| = \mathcal{O}(|r_2|)$.

  Thus, by definition, $w_1 \cdot w_2 \in L(r, \nu)$ and $|w_1 \cdot w_2| = \mathcal{O}(|r|)$.
- Case 3: $r = (r_1)^*$.
  This case is trivial since $\varepsilon \in L(r, \nu)$.
- Case 4: $r = a \downarrow_{x_i} .\{r_1\}$.
  If $L(r, \nu) \neq \emptyset$, then by the definition, $L(r_1, \nu[x_i \leftarrow d])$ is not empty, for some data value d. By the induction hypothesis, there exists $w_1 \in L(r_1, \nu[x_i \leftarrow d])$ such that $|w_1| = \mathcal{O}(|r_1|)$. By definition, $\binom{a}{d} w_1 \in L(r, \nu)$.

This completes the proof of Proposition 5.12. □

The NP membership now follows from Proposition 5.12, where given a REWB r, we simply guess a data word $w \in L(r)$ of length $\mathcal{O}(|r|)$. The verification that $w \in L(r)$ can also be done in NP (this follows from Theorem 4.5 and the fact that each REWB can be converted into an equivalent RA in polynomial time).

Note that the data values here can be made small as well. This follows from the fact that in a word accepted by a register automaton one can replace the data values with the ones from the set $1, \ldots k + 1$, where k is the number of registers (see Lemma 3.1), while retaining the acceptance condition. Thus we can always assume that the values appearing in our word are not bigger than the number of variables in our expression plus one.

We prove NP hardness via a reduction from 3-SAT.

Assume that $\varphi = (\ell_{1,1} \vee \ell_{1,2} \vee \ell_{1,3}) \wedge \cdots \wedge (\ell_{n,1} \vee \ell_{n,2} \vee \ell_{n,3})$ is the given 3-CNF formula, where each $\ell_{i,j}$ is a literal. Let $x_1, \ldots x_k$ denote the variables occurring in φ. We say that the literal $\ell_{i,j}$ is negative, if it is a negation of a variable. Otherwise, we call it a positive literal.

We will define a closed REWB r over $\Sigma[y_1, z_1, y_2, z_2, \ldots, y_k, z_k]$ of length $\mathcal{O}(n)$ such that φ is satisfiable if and only if $L(r) \neq \emptyset$.

Let r be the following REWB.

$$\begin{aligned} r \;:=\; & a \downarrow_{y_1} .\{a \downarrow_{z_1} .\{a \downarrow_{y_2} .\{a \downarrow_{z_2} .\{\cdots\{a \downarrow_{y_k} .\{a \downarrow_{z_k} .\{ \\ & (r_{1,1} + r_{1,2} + r_{1,3}) \cdots (r_{n,1} + r_{n,2} + r_{n,3})\}\} \ldots\}, \end{aligned}$$

$$r_{i,j} \;:=\; \begin{cases} b[y_k^{=} \wedge z_k^{=}] & \text{if } \ell_{i,j} = x_k \\ \\ b[y_k^{=} \wedge z_k^{\neq}] + b[z_k^{=} \wedge y_k^{\neq}] & \text{if } \ell_{i,j} = \neg x_k \end{cases}$$

Obviously, $|r| = \mathcal{O}(n)$. We are going to prove that φ is satisfiable if and only if $L(r) \neq \emptyset$.

Assume first that φ is satisfiable. Then there is an assignment $f : \{x_1, \ldots, x_k\} \mapsto \{0, 1\}$ making φ true. We define the evaluation $\nu : \{y_1, z_1, \ldots y_n, z_n\} \mapsto \{0, 1\}$ as follows.

- If $f(x_i) = 1$, then $\nu(y_i) = \nu(z_i) = 1$.
- If $f(x_i) = 0$, then $\nu(y_i) = 0$ and $\nu(z_i) = 1$.

We define the following data word.

$$w \;:=\; \binom{a}{\nu(y_1)}\binom{a}{\nu(z_1)} \cdots \binom{a}{\nu(y_k)}\binom{a}{\nu(z_k)} \underbrace{\binom{b}{1} \cdots \binom{b}{1}}_{n \text{ times}}$$

To see that $w \in L(r)$, we observe that the first $2k$ labels are parsed to bind values $y_1, z_1, \ldots y_k, z_k$ to corresponding values determined by ν. To parse the remaining $\binom{b}{1} \cdots \binom{b}{1}$, we observe that for each $i \in \{1, \ldots, n\}$, $\ell_{i,1} \vee \ell_{i,2} \vee \ell_{i,3}$ is true according to the assignment f if and only if $\binom{b}{1} \in L(r_{i,1} + r_{i,2} + r_{i,3}, \nu)$.

Conversely, assume that $L(r) \neq \emptyset$. Let

$$w = \binom{a}{d_{y_1}}\binom{a}{d_{z_1}} \cdots \binom{a}{d_{y_k}}\binom{a}{d_{z_k}}\binom{b}{d_1} \cdots \binom{b}{d_n} \in L(r).$$

We define the following assignment $f : \{x_1, \ldots, x_k\} \mapsto \{0, 1\}$.

$$f(x_i) \quad = \quad \begin{cases} 1 & \text{if } d_{y_i} = d_{z_i} \\ 0 & \text{if } d_{y_i} \neq d_{z_i} \end{cases}$$

We are going to show that f is a satisfying assignment for φ. Now since $w \in L(r)$, we have

$$\binom{b}{d_1} \cdots \binom{b}{d_n} \quad \in \quad L((r_{1,1} + r_{1,2} + r_{1,3}) \cdots (r_{n,1} + r_{n,2} + r_{n,3}), \nu),$$

where $\nu(y_i) = d_{y_i}$ and $\nu(z_i) = d_{z_i}$. In particular, we have for every $j = 1, \ldots, n$,

$$\binom{b}{d_j} \quad \in \quad L(r_{j,1} + r_{j,2} + r_{j,3}, \nu).$$

W.l.o.g, assume that $\binom{b}{d_j} \in L(r_{j,1})$. There are two cases.

- If $r_{j,1} = b[y_i^{=} \wedge z_i^{=}]$, then by definition, $\ell_{j,1} = x_i$, hence the clause $\ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3}$ is true under the assignment f.
- If $r_{j,1} = b[y_i^{=} \wedge z_i^{\neq}] + b[z_i^{=} \wedge y_i^{\neq}]$, then by definition, $\ell_{j,1} = \neg x_i$, hence the clause $\ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3}$ is true under the assignment f.

Thus, the assignment f is a satisfying assignment for the formula φ. This completes the proof of NP-hardness.

5.3. Proof of the undecidability of the universality problem for REWB

We are first going to prove that given an REWB r over $\Sigma[x_1, \ldots, x_k]$, checking whether $L(e) = (\Sigma \times \mathcal{D})^*$ is undecidable. This immediately implies that given r_1, r_2, checking whether $L(r_1) \subseteq L(r_2)$ is undecidable.

The proof is similar to the proof of the universality of register automata in [9]. The reduction is via Post Correspondence Problem (PCP), which is defined as follows. An instance of PCP is a set of of pair of strings

$$I = \{(u_1, v_1), \ldots, (u_n, v_n)\},$$

where $u_i, v_i \in \Sigma^*$. A solution of the instance I is a sequence $l_1, \ldots, l_m$ such that $u_{l_1} \cdots u_{l_m} = v_{l_1} \cdots v_{l_m}$.

Let $\$, \#$ be two special symbols not in Σ. Now a solution $l_1, \ldots, l_m$ of the PCP instance I can be encoded into data word $w_1 \binom{\#}{h} w_2$ over $\Sigma \cup \{\$, \#\}$, where

$$w_1 = \binom{\$}{e_1}\binom{a_1}{d_1}\cdots\binom{a_{\ell_1}}{d_{\ell_1}}\binom{\$}{e_2}\binom{a_{\ell_1+1}}{d_{\ell_1+1}}\cdots\binom{a_{\ell_1+\ell_2}}{d_{\ell_1+\ell_2}}\binom{\$}{e_3}\cdots\cdots\binom{\$}{e_m}\binom{a_{\ell_1+\cdots+\ell_{m-1}}}{d_{\ell_1+\cdots+\ell_{m-1}}}\cdots\binom{a_\ell}{d_\ell}$$

$$w_2 = \binom{\$}{g_1}\binom{b_1}{f_1}\cdots\binom{b_{\ell_1}}{f_{\ell_1}}\binom{\$}{g_2}\binom{b_{\ell_1+1}}{f_{\ell_1+1}}\cdots\binom{b_{\ell_1+\ell_2}}{f_{\ell_1+\ell_2}}\binom{\$}{g_3}\cdots\cdots\binom{\$}{g_m}\binom{b_{\ell_1+\cdots+\ell_{m-1}}}{f_{\ell_1+\cdots+\ell_{m-1}}}\cdots\binom{b_\ell}{f_\ell}$$

where $\ell = \ell_1 + \ell_2 + \cdots + \ell_m$, and

(C1) The symbol $\#$ appears only once.

(C2) $\mathsf{Proj}_\Sigma(w_1) \in (\$ \cdot u_1 + \cdots + \$ \cdot u_n)^*$.

(C3) $\mathsf{Proj}_\Sigma(w_2) \in (\$ \cdot v_1 + \cdots + \$ \cdot v_n)^*$.

(C4) The data values e_i's and d_i's are pairwise different.

(C5) The data values g_i's and f_i's are pairwise different.

(C6) $e_1 = g_1$ and $e_m = g_m$.

(C7) $d_1 = f_1$ and $d_\ell = f_\ell$.

(C8) For all $i \in \{1, \ldots, m-1\}$, there exists $j \in \{1, \ldots, m-1\}$ such that $e_i = g_j$ and $e_{i+1} = g_{j+1}$.

(C9) For all $i \in \{1, \ldots, \ell-1\}$, there exists $j \in \{1, \ldots, \ell-1\}$ such that $d_i = f_j$ and $d_{i+1} = f_{j+1}$.

(C10) For all $i, j \in \{1, \ldots, \ell\}$, if $d_i = f_j$, then $a_i = b_j$.

(C11) For all $i, j \in \{1, \ldots, m\}$, if $e_i = g_j$, then the pair of strings $(a_{\ell_1+\ldots+\ell_{i-1}+1} \cdots a_{\ell_1+\ldots+\ell_i}, b_{\ell_1+\ldots+\ell_{j-1}+1} \cdots b_{\ell_1+\ldots+\ell_j}) \in I$.

Now it is straightforward to show that there exists a solution to the PCP instance I if and only if there exists a data word over $\Sigma \cup \{\$, \#\}$ that satisfies Conditions (C1)–(C11) above.

We now prove that we get undecidability even when using expressions with only one variable. Let r be an REWB over $\Sigma[x]$.

As above, let $\$, \#$ be two special symbols not in Σ and define $\Gamma = \Sigma \cup \{\$, \#\}$. Now a solution $l_1, \ldots, l_m$ of the PCP instance I can be encoded into data word $w_1 \binom{\#}{h} \mathrm{REV}(w_2)$ over $\Sigma \cup \{\$, \#\}$, where w_1, w_2 are defined as above and $\mathrm{REV}(w_2)$ is the reversal of w_2.

We then construct an REWB r over $\Gamma[x]$ that accepts a data word $w = w_1 \# \mathrm{REV}(w_2)$ such that $w_1 \# w_2$ does not satisfies at least one of the Conditions (C1) to (C11) above. The REWB r is obtained by taking the union of the following.

- The negations of each (C1), (C2), (C3) which can be written in a standard regular expression without variables.
- The negation of (C4) which can be written as:

$$\Big(\Gamma^*\$ \downarrow_x .\{\Gamma^*\$[x^{=}]\} \quad + \quad \Gamma^* \bigcup_{a\in\Sigma} \big(a \downarrow_x .\{\Gamma^* a[x^{=}]\}\big)\Big)\#\Gamma^*$$

  The negation of (C5) can be written in a similar manner.
- The negation of (C6) which can be written as:

$$\$ \downarrow_x .\{\Gamma^* \cdot \$[x^{\neq}]\} \quad + \quad \Gamma^*\$ \downarrow_x .\{\# \cdot \Sigma^*\$[x^{\neq}]\}\Gamma^*.$$

  The negation of (C7) can be written in a similar manner.
- The negation of (C8) which can be written as:

$$\Gamma^*\$ \downarrow_x .\left\{\Gamma^*\#(\$[x^{\neq}]\Sigma)^* + \Sigma^*\$ \downarrow_x .\{\Gamma * \#\Gamma * \$[x^{\neq}]\}\Sigma^*\$[x^{=}]\right\}.$$

  Note that here we use the fact that (C8) can be paraphrased as follows:

  1. For all $i \in \{1,\ldots,m-1\}$ exists $j \in \{1,\ldots,m-1\}$ such that $e_i = g_j$
  2. For all $i \in \{1,\ldots,m-1\}$ and for all $j \in \{1,\ldots,m-1\}$ if $e_i = g_j$ then $e_{i+1} = g_{j+1}$.

  (Recall that by (C6) we have that $e_1 = g_1$.)

  The negation of (C9) can be written in a similar manner.
- The negation of (C10) and the negation of (C11), which can be written in a straightforward manner using only one variable.

It is straightforward to see that the PCP instance I has no solution if and only if $L(r) = (\Sigma_1 \times \mathcal{D})^*$. This concludes our proof of Theorem 5.5.

6. Regular expressions with equality

In this section we define yet another kind of expressions, regular expressions with equality, that will have significantly better algorithmic properties that regular expressions with memory or binding, while still retaining much of their expressive power. The idea is to allow checking for (in)equality of data values at the beginning and at the end of subwords conforming to subexpressions and not by using variables.

Definition 6.1 (Expressions with equality)**.** *Let Σ be a finite alphabet. The* regular expressions with equality (REWE) *are defined by the grammar:*

$$e \;:=\; \emptyset \mid \varepsilon \mid a \mid e + e \mid e \cdot e \mid e^{+} \mid e_{=} \mid e_{\neq}$$

where $a \in \Sigma$.

The language $L(e)$ of data words denoted by a regular expression with equality e is defined as follows.

- $L(\emptyset) = \emptyset$.
- $L(\varepsilon) = \{\varepsilon\}$.
- $L(a) = \{\binom{a}{d} \mid d \in \mathcal{D}\}$.
- $L(e \cdot e') = L(e) \cdot L(e')$.
- $L(e + e') = L(e) \cup L(e')$.
- $L(e^{+}) = \{w_1 \cdots w_k \mid k \geq 1 \text{ and each } w_i \in L(e)\}$.
- $L(e_{=}) = \{\binom{a_1}{d_1} \ldots \binom{a_n}{d_n} \in L(e) \mid n \geq 2 \text{ and } d_1 = d_n\}$.
- $L(e_{\neq}) = \{\binom{a_1}{d_1} \ldots \binom{a_n}{d_n} \in L(e) \mid n \geq 2 \text{ and } d_1 \neq d_n\}$.

Without any syntactic restrictions, there may be "pathological" expressions that, while formally defining the empty language, should nonetheless be excluded as really not making sense. For example, $\varepsilon_{=}$ is formally an expression, and so is $a_{\neq}$, although it is clear they cannot denote any data word. We exclude them by defining well-formed expressions as follows. We say that the usual regular expression e reduces to ε (respectively, to singletons) if $L(e)$ is ε or $\emptyset$ (or $|w| \leq 1$ for all $w \in L(e)$). Then we say that regular expression with equality is *well-formed* if it contains no subexpressions of the form $e_{=}$ or $e_{\neq}$, where e reduces to ε, or to singletons. From now on we will assume that all our expressions are well formed.

Note that we use $^+$ instead of $*$ for iteration. This is done for technical purposes (the ease of translation) and does not reduce expressiveness, since we can always use e^* as shorthand for $e^+ + \varepsilon$.

Example 6.2. *Consider the REWE $e_1 = \Sigma^* \cdot (\Sigma \cdot \Sigma^+)_= \cdot \Sigma^*$. The language $L(e_1)$ consists of the data words that contain two different positions with the same data value. The REWE $e_2 = (\Sigma \cdot \Sigma^+)_{\neq}$ denotes the language of data words in which the first and the last data value are different.*

By straightforward induction, we can easily convert an REWE to its equivalent REWB. The following theorem states that REWE is strictly weaker than REWB. The proof can be found in Section 6.1.

Theorem 6.3. *In terms of expressive power REWB are strictly more expressive than REWE.*

Closure properties. As immediately follows from their definition, languages denoted by regular expressions with equality are closed under union, concatenation, and Kleene star. Also, it is straightforward to see that they are closed under automorphisms. However:

Proposition 6.4. *The REWE languages are not closed under intersection and complement.*

Proof. Observe first that the expression $\Sigma^* \cdot (\Sigma \cdot \Sigma^+)_= \cdot \Sigma^*$ defines the language of data words containing two positions with the same data value. The complement of this language is the set of all data words where all data values are different, which is not recognizable by register automata [1]. Since REWE are strictly contained in REWB, which are in turn contained in register automata, the non closure under complement follows.

To see that the REWE languages are not closed under intersection, we first show that the language

$$L = \left\{ \binom{a}{d_1}\binom{a}{d_2}\binom{a}{d_3} \;\middle|\; d_1 \neq d_2, d_1 \neq d_3 \text{ and } d_2 \neq d_3 \right\}$$

is not recognizable by any regular expression with equality. To prove this we simply try out all possible combinations of expressions that use at most three concatenated occurrences of a. Note that we can eliminate any expression with more that three as, or one that uses * (since this results in arbitrary long words),

or union (since every member of the union would have to define words from this language and since we do not use constants we cannot just split the language into two or more parts). Also, no $=$ can occur in our expression (for subexpressions of length at least 2). This reduces the number of potential expressions to denote the language to finitely many possibilities, and we simply try them all.

Now observe that the expression $e_1 = ((a \cdot a)_{\neq} \cdot a)_{\neq}$ defines the language

$$L_1 = \left\{ \binom{a}{d_1}\binom{a}{d_2}\binom{a}{d_3} \;\middle|\; d_1 \neq d_2 \text{ and } d_1 \neq d_3 \right\}.$$

Similarly $e_2 = a \cdot (a \cdot a)_{\neq}$ defines

$$L_2 = \left\{ \binom{a}{d_1}\binom{a}{d_2}\binom{a}{d_3} \;\middle|\; d_2 \neq d_3 \right\}.$$

Note that $L = L_1 \cap L_2$, so if regular expressions with equality were closed under intersection they would also have been able to define the language L. □

Decision problems. Next, we show that nonemptiness and membership problems for REWE can be decided in PTIME. To obtain a PTIME algorithm we first show how to reduce REWE to pushdown automata when only finite alphabets are involved. Assume that we have a finite set D of data values. We can then inductively construct PDAs $P_{e,D}$ for all regular expressions with equality e. The words recognized by these automata will be precisely the words from $L(e)$ whose data values come from D. Formally, we have the following.

Lemma 6.5. *Let D be a finite set of data values. For any regular expression with equality e we can construct a PDA $P_{e,D}$ such that the language of words accepted by $P_{e,D}$ is equal to the set of data words in $L(e)$ whose data values come from D. Moreover, the PDA $P_{e,D}$ has at most $O(|e|)$ states and $O(|e| \times (|D|^2 + |e|))$ transitions, and can be constructed in polynomial time.*

The proof of Lemma 6.5 can be found in Section 6.2. Using this result we can show the PTIME bound.

Theorem 6.6. *The nonemptiness problem and the membership problem for REWE are in* PTIME.

Proof. For nonemptiness, let e be the given REWE. By straightforward induction on the length of e, we can show that if $L(e) \neq \emptyset$, then there exists a data word w

in which there are at most n different data values, where n is the number of times $=$ and $\neq$ appear in e. Now, we construct the PDA $P_{e,D}$, where $D = \{0, 1, \ldots, n+1\}$ in time polynomial in the length of e. Since the nonemptiness of PDA can be checked in PTIME, we obtain the PTIME upper bound for the the emptiness problem for REWE.

Next we prove that membership can also be decided in PTIME. As in the nonemptiness problem, we construct a PDA $P_{e,D}$ for e and $D = \{0, 1, \ldots, n\}$, where n is the length of the input word w. Furthermore, we can assume that data values in w come from the set D. Next we simply check that the word is accepted by $P_{e,D}$ and since this can be done in PTIME, we get the desired result. □

PDAs vs NFAs. At this point the reader may ask whether it could have been possible to use NFA instead of PDA in our proof above. We remark that yes, it could have been possible to use NFA instead of PDA, but it comes with an exponential blow-up, as stated in the following proposition.

Proposition 6.7. *For every regular expression with equality e over the alphabet Σ and a finite set D of data values there exists an NFA $\mathcal{A}_{e,D}$, of size exponential in $|e|$, recognizing precisely those data words from $L(e)$ that use data values from D.*

Proof. We prove this by structural induction on regular expressions with equality. All of the standard cases are carried out as usual. Thus we only have to describe the construction for subexpressions of the form $e_=$ and $e_{\neq}$. In both cases by the induction hypothesis we know that there is an NFA $\mathcal{A}_{e,D}$ recognizing words in $L(e)$ with data values from D. The automaton for $\mathcal{A}_{e_{\neq},D}$ (and likewise for $\mathcal{A}_{e_=,D}$) will consist of $|D|$ disjoint copies of $\mathcal{A}_{e,D}$, each designated to remember the first data value read when processing the input. According to this, whenever our automaton would enter a final state we test that the current data value is different (or the same) to the one corresponding to this copy of the original automaton. □

However, the exponential lower bound is the best we can do in the general case. To see this, we define a sequence of regular expressions with equality $\{e_n\}_{n\in\mathbb{N}}$, over the alphabet $\Sigma = \{a\}$, and each of length linear in n. We then show that for $D = \{0, 1\}$ every NFA over the alphabet $\Sigma \times D$ recognizing precisely those data words from $L(e_n)$ with data values in D has length exponential in $|e_n|$.

To prove this we will use the following theorem for proving lower bounds of NFAs [26]. Let $L \subseteq \Sigma^*$ be a regular language and suppose there exists a set $P = \{(x_i, y_i) : 1 \leq i \leq n\}$ of pairs such that:

1. $x_i \cdot y_i \in L$, for every $i = 1, \ldots n$, and

2. $x_i \cdot y_j \notin L$, for $1 \leq i, j \leq n$ and $i \neq j$.

Then any NFA accepting L has at least n states.

Thus to prove our claim it suffices to find such a set of size exponential in the length of e_n.

Next we define the expressions e_n inductively as follows:

- $e_1 = (a \cdot a)_=$,
- $e_{n+1} = (a \cdot e_n \cdot a)_=$.

It is easy to check that $L(e_n) = \{w \cdot w^{-1} : w \in (\Sigma \times \{0,1\})^n\}$, where w^{-1} denotes the reverse of w.

Now let $w_1, \ldots w_{2^n}$ be a list of all the elements in $(\Sigma \times \{0,1\})^n$ in arbitrary order. We define the pairs in P as follows:

- $x_i = w_i$,
- $y_i = (w_i)^{-1}$.

Since these pairs satisfy the above assumptions 1) and 2), we conclude, using the result of [26], that any NFA recognizing $L(e_n)$ has at least $O(2^{|e_n|})$ states.

6.1. Separating REWB from REWE

Consider the following REWB $r = a \downarrow_x .\{(a[x^{\neq}])^*\}$. In the rest of this section, we are going to show that there is no REWE that expresses the language $L(r)$.

To prove this we introduce a new kind of automata, called *weak register automata*, show that they subsume regular expressions with equality and that they can not express the language $a \downarrow_x .\{(a[x^{\neq}])^*\}$ of a-labeled data words in which all data values are different from the first one.

The main idea behind weak register automata is that they erase the data value that was stored in the register once they make a comparison, thus rendering the register empty. We denote this by putting a special symbol $\perp$ from $\mathcal{D}$ in the register. Since they have a finite number of registers, they can keep track of only finitely many positions in the future, so in the case of our language, they can only check that a fixed finite number of data values is different from the first one. We proceed with formal definitions.

The definition of weak k-register automaton is similar to Definition 2.1. The only explicit change we make is that the set of transitions T contains transitions of the form $(q, a, \varphi) \to (I, q')$, where q and q' are states, a is a letter from Σ, φ a condition and $I \subseteq \{x_1, \ldots, x_k\}$ is a set of registers.

Definition of configuration remains the same as before, but the way we move from one configuration to another changes.

From a configuration $c = (q, \tau)$ we can move to a configuration $c' = (q', \tau')$ while reading the position j in the data word $w = \binom{a_1}{d_1} \cdots \binom{a_n}{d_n}$ if the following holds:

- $(q, a_j, \varphi) \to (I, q')$ is a transition in $\mathcal{A}$,
- $d_j, \tau \models \varphi$ and
- τ' coincides with τ except that every register mentioned in φ is set to be empty (i.e. to contain $\perp$) and the ith component of τ' is set to d whenever $x_i \in I$.

The last item simply tells us that if we used a condition like $c = x_3^{=} \wedge x_7^{\neq}$ in our transition, we would afterwards erase data values that were stored in registers 3 and 7. Note that we can immediately rewrite these registers with the current data value. The ability to store a data value into more than one register is needed to simulate expressions of the form $((aa)_{=}a)_{\neq}$, where the first data value is used twice.

The notion of acceptance and an accepting run is the same as before.

We now show that weak register automata can not recognize the language L of all data words where first data value is different from all other data values, i.e. the language denoted by the expression $a \downarrow_x .\{(a[x^{\neq}])^*\}$.

Assume to the contrary, that there is some weak $k-$register data word automaton $\mathcal{A}$ recognizing L. Since data word $w = \binom{a}{d_1}\binom{a}{d_2} \cdots \binom{a}{d_k}\binom{a}{d_{k+1}}\binom{a}{d_{k+2}}$, where d_is are pairwise different is in L, there is an accepting run of $\mathcal{A}$ on w. The idea behind the proof is that $\mathcal{A}$ can check that only the first $k+1$ positions have different data value from the first.

First we note a few things. Since every data value in the word w is different, no $=$ comparisons can be used in conditions appearing in this run (otherwise the condition test would fail and the automaton would not accept).

Now note that since we have only k registers, and with every comparison we empty the corresponding registers one of the following must occur:

- There is a data value $1 < i < k + 2$ such that the condition used when processing this data value is $\mathtt{tt}$. In this case we simply replace d_i by d_1 and get an accepting run on a word that has the first data value repeated – a contradiction. Note that we could store d_i in that transition, but since afterwards we only test for inequality this will not alter the rest of the computation.
- There is a data value such that when the automaton reads it it does not use any register with the first data value, i.e. d_1, stored. Note that this must happen, because at best we can store the first data value in all the registers at the beginning of our run, but after that each time we read a data value and compare it to the first we lose the first data value in this register. But then again we can simply replace this data value with d_1 and get an accepting run (just as before, if this data value gets stored in this transition and then used later it can only be used in a $\neq$ comparison, which is also true for d_1, so the run remains accepting). Again we arrive at a contradiction.

This shows that no weak register automaton can recognize the language L.
To complete the proof we still have to show the following:

Lemma 6.8. *For every regular expression with equality e there exists a weak k-register automaton $\mathcal{A}_e$, recognizing the same language of data words, where k is the number of times $=, \neq$ symbols appear in e.*

The proof can be done by a straightforward induction on the structure of e.

6.2. Proof of Lemma 6.5

First we provide some intuition behind the construction. It is quite clear how to construct the automata for $e = \varepsilon, e = \emptyset$ and $e = a$. For $e_1 + e_2, e_1 \cdot e_2$ and e_1^+ we use standard constructions, while for $e = (e_1)_=$, or $e = (e_1)_\neq$ we push the first data value on the stack, mark it by a new stack symbol and then proceed with the run of the automaton for e_1 which exists by the induction hypothesis. Every time we enter a final state of that automaton we simply empty the stack until we reach the first data value (here we use the new stack symbol) and compare it for equality or inequality with the last data value of the input word.

Next we prove the lemma formally.

We will assume that we do not use expressions $e = \varepsilon$ and $e = \emptyset$ to avoid some technical problems. Note that this is not a problem since we can always detect the presence of these expressions in the language in linear time and code them into our automata by hand.

In what follows we will use the notational conventions about PDAs from [27].

We construct our PDAs so that they accept by final state and furthermore have the property that only transitions of the kind $(q_0, \binom{a}{d}, X, \alpha, q)$ leave the initial state (that is any transition leaving the initial state will consume a letter) and every transition entering a final state will consume a letter. We will maintain these properties throughout the inductive construction.

Assume now that we are given a well-formed regular expression with equality e (with no subexpressions of the form ε and $\emptyset$) over the alphabet Σ and a finite set of data values D. We construct, by induction on e, a PDA $P_{e,D}$ over the alphabet $\Sigma \times D$ such that:

- $w = \binom{a_1}{d_1} \ldots \binom{a_n}{d_n}$ is accepted by $P_{e,D}$ if and only if $w \in L(e)$ and $d_1, \ldots, d_n \in D$.
- There are no ε-transitions leaving the initial state (that is every transition from the initial state will consume a symbol).
- There is no ε-transition entering a final state.

We note that our PDAs will accept by final state and use start stack symbol.

- If $e = a$, with $a \in \Sigma$ we define $P_{e,D} = (Q, q_0, \Sigma', \Gamma, Z_0, F, \delta)$, where:
  - $Q = \{q_0, q_1\}$,
  - $F = \{q_1\}$,
  - $\Sigma' = \Sigma \times D$,
  - $\Gamma = D \cup \{Z_0\}$, and
  - $\delta(q_0, \binom{a}{d}, Z_0) = \{(q_1, \varepsilon)\}$, for every $d \in D$.

  It is straightforward to check that $P_{e,D}$ has the desired properties.
- Cases $e = e_1 + e_2$ and $e = e_1 \cdot e_2$ and $e = e_1^+$ are straightforward and are executed in a standard way using the inductive assumption to avoid ε-transitions from initial state and to final states.
- If $e = (e_1)_=$ then let $P_{e_1,D} = \{Q, q_0, \Sigma', \Gamma, Z_0, F, \delta\}$ be the PDA for e_1 and D which exists by the inductive hypothesis.

  We define $P_{e,D} = (Q', q_0, \Sigma', \Gamma', Z_0', F', \delta')$, where:

- $Q' = Q \cup \{q', q'', q_f, q'_f, q''_f\}$,
- $F' = \{q_f\}$,
- $\Gamma' = \Gamma \cup \{X_0\}$, where X_0 is a new stack symbol and
- To δ' we add all the transition from δ, plus
  1. For every $(q_0, \binom{a}{d}, Z_0) \to (q_1, \alpha)$ in δ we add the transitions:
     (a) $(q_0, \binom{a}{d}, Z'_0) \to (q', dZ'_0)$,
     (b) $(q', \varepsilon, d) \to (q'', X_0 d)$,
     (c) $(q'', \varepsilon, X_0) \to (q'', Z_0 X_0)$, and
     (d) $(q'', \varepsilon, Z_0) \to (q_1, \alpha)$ to δ'.
  2. For every $(q'_j, \binom{a}{d}, X) \to (q_j, \alpha)$ in δ, with $q_j \in F$ we add:
     (a) $(q'_j, \varepsilon, X) \to (q'_f, \alpha)$,
     (b) $(q'_f, \varepsilon, Y) \to (q'_f, \varepsilon)$, for every $Y \in \Gamma$,
     (c) $(q'_f, \varepsilon, X_0) \to (q''_f, \varepsilon)$, and
     (d) $(q''_f, \binom{a}{d}, d) \to (q_f, \varepsilon)$ to δ'.

Note first that q_1 in the first item of transitions added to δ' will never be a final state and that q'_j in the second item will never be the initial state. This simply follows from the assumption that our expressions are well-formed. Furthermore it is easy to see that no ε-transitions leave the initial state or enter a final state in our automaton.

Next we show that the constructed automaton recognizes the language $L(e)$ restricted to data values in D. To see this note that the first block of newly added transitions simply pushes the first data value onto the stack, covers it with the new stack symbol X_0, and then proceeds as $P_{e_1,D}$ would right until the point when $P_{e_1,D}$ enters a final state. At this point $P_{e,D}$ starts to empty the stack until it sees the new symbol X_0. After popping this symbol we know that the first data value is written below it, so we compare it with the current data value for equality. If they are equal we proceed to the final state and accept (provided we have reached the end of the word).

Note that this proves that every word accepted by $P_{e,D}$ is a word accepted by $P_{e_1,D}$ that has equal first and last data value and is thus in $L(e)$ by the inductive hypothesis. The converse follows easily from this same observation and the induction hypothesis.

Note also that we can not accept any word that does not use the first transition that stores the first data value onto the stack simply because we will not have it on the stack (below X_0) when we want to proceed to the final state.

- If $e = (e_1)_{\neq}$ then let $P_{e,D}$ will be the same as for $(e_1)_{=}$, except that 2(d) changes to $(q''_f, \binom{a}{d}, d') \to (q_f, \varepsilon)$, for all $d' \neq d$ in D. The proof that this is correct is identical as in that case.

Note that the size of the stack alphabet is at most $|D| + 2|e|$, since we have to add a new stack symbol for every $=, \neq$ that appears in e (as well as the new initial stack symbol).

To see that the automaton is linear in the length of expression note that we only add new states when constructing automaton for $(e_1)_{=}, (e_1)_{\neq}$ and $e_1 + e_2$. In each case we add only a fixed number of states (five in the first two cases and one in the last).

To count the number of transitions observe that we add at most $|D|^2+|D|+|e|$ transitions between any two states when we construct the automaton for $(e_1)_{\neq}$ (all other cases have $|D|$, or $|e|$ transitions or less). Thus we have at most $O(|e| \times (|D|^2 + |e|))$ transitions in our automaton.

7. Summary and conclusions

Motivated by the need for concise representation of data word languages we defined and studied several classes of expressions for describing them. As our base model we took register automata, a common formalism for specifying languages of data words and, in addition to defining a class of expressions capturing register automata, we also looked into several subclasses that allow for more efficient algorithms for main reasoning tasks while at the same time retaining enough expressive power to be of interest in applications such as querying data graphs [19], or modelling infinite-state systems with finite control [13].

Here we would also like to note that yet another model of automata for data words exists, namely *variable automata*, or VFA for short [14]. Although originally defined to operate over infinite alphabets it is easy to extend them to data words [28]. In this paper we did not consider variable automata as our goal was to study regular formalisms for data words and VFA exhibit somewhat irregular behaviour by allowing the usage of a free variable that acts globally. Moreover, all of the complexity bounds and closure properties of VFA were already established in [14] and they readily extend to the setting of data words. For completeness we

summarise the results on VFA below and discuss how they compare to formalisms introduced in this paper.

The first class of problems we studied was the complexity/decidability of nonemptiness, membership and universality. We showed that with the increase of expressive power of a language the complexity of the first two problems rises correspondingly. With universality the situation is different, as all of the languages have undecidable universality problem. As the unusual nature of VFA makes them orthogonal to languages we introduced this is also reflected on decision problems – namely, VFA are the only formalism having a higher bound for membership than for nonemptiness. The summary of the complexity bounds is presented in Table 1.

	RA	REM	REWB	REWE	VFA
nonemptiness	PSPACE-c	PSPACE-c	NP-c	PTIME	NLOGSPACE-c*
membership	NP-c	NP-c	in NP	PTIME	NP-c*
universality	undecidable	undecidable	undecidable	undecidable†	undecidable*

Table 1: Complexity of main decision problems. Results marked by * are from [14], while † was first shown in [24] by extending the proof of Theorem 5.5

As is common in language theory, we also studied basic closure properties of our languages. A summary of the results is given in Table 2. We can see that while all of the formalisms are closed under union, concatenation and Kleene star, none is closed under complementation. We also studied closure under intersection, and while most languages do enjoy this property (due to a fact that one can carry out the standard NFA product construction), for the case of REWB and REWE we can show that this is no longer true.

	RA	REM	REWB	REWE	VFA*
union	+	+	+	+	+
intersection	+	+	−	−	+
concatenation	+	+	+	+	+
Kleene star	+	+	+	+	+
complement	−	−	−	−	−

Table 2: Closure properties of data word defining formalisms. Results on VFA follow from [14]

Lastly, we also studied how the five classes of languages compare one to another. While regular expressions with memory were originally introduced as an

expression analogue of register automata, here we also showed that they subsume expressions with binding as well as expressions with equality. VFA, on the other hand, are orthogonal to all the other formalisms studied in this paper, as they can express properties out of the reach of register automata, while failing to capture even REWE. For example they can state that all data values differ from the last value in the word – a property not expressible by RA; while at the same time they can not capture the language defined by the REWE $((aa)_{=})^{+}$ (see [14] for details). We thus obtain:

Theorem 7.1. *The following relations hold, where $\subsetneq$ denotes that every language defined by formalism on the left is definable by the formalism on the right, but not vice versa.*

- *REWE $\subsetneq$ REWB $\subsetneq$ REM = register automata.*
- *VFA are incomparable in terms of expressive power with REWE, REWB, REM and register automata.*

[1] M. Kaminski, N. Francez, Finite memory automata, Theoretical Computer Science 134 (1994) 329–363.

[2] L. Libkin, Logics for unranked trees: an overview, Logical Methods in Computer Science 2 (2006).

[3] M. Marx, Conditional xpath, ACM Trans. Database Syst. 30 (2005) 929–959.

[4] T. Schwentick, Automata for xml – a survey, JCSS 73 (2007) 289–315.

[5] M. Bojanczyk, P. Parys, Xpath evaluation in linear time, J. ACM 58 (2011).

[6] M. Bojanczyk, S. Lasota, An extension of data automata that captures xpath, in: 25th Annual IEEE Symposium on Logic in Computer Science (LICS), 2010, pp. 243–252.

[7] M. Bojanczyk, C. David, A. Muscholl, T. Schwentick, L. Segoufin, Two-variable logic on words with data, ACM TOCL 12 (2011).

[8] D. Figueira, Satisfiability of downward xpath with data equality tests, in: 28th ACM Symposium on Principles of Database Systems (PODS), 2009, pp. 197–206.

[9] F. Neven, T. Schwentick, V. Vianu, Finite state machines for strings over infinite alphabets, ACM Trans. Comput. Log. 5 (2004) 403–435.

[10] L. Segoufin, Automata and logics for words and trees over an infinite alphabet, in: CSL, 2006, pp. 41–57.

[11] M. Benedikt, C. Ley, G. Puppis, Automata vs. logics on data words, in: CSL, 2010, pp. 110–124.

[12] T. Colcombet, C. Ley, G. Puppis, On the use of guards for logics with data, in: MFCS, 2011, pp. 243–255.

[13] S. Demri, R. Lazić, Ltl with the freeze quantifier and register automata, ACM TOCL 10 (2009).

[14] O. Grumberg, O. Kupferman, S. Sheinvald, Variable automata over infinite alphabets, in: Proceedings of the 4th International Conference on Language and Automata Theory and Applications (LATA), 2010, pp. 561–572.

[15] R. Angles, C. Gutierrez, Survey of graph database models, ACM Computing Surveys 40 (2008).

[16] A. Mendelzon, P. Wood, Finding regular simple paths in graph databases, SIAM Journal on Computing 24 (1995) 1235–1258.

[17] D. Calvanese, G. De Giacomo, M. Lenzerini, M. Vardi, Rewriting of regular expressions and regular path queries, Journal of Computer and System Sciences 64 (2002) 443–465.

[18] P. Barceló, C. Hurtado, L. Libkin, P. Wood, Expressive languages for path queries over graph-structured data, in: 29th ACM Symposium on Principles of Database Systems (PODS), 2010, pp. 3–14.

[19] L. Libkin, D. Vrgoč, Regular Path Queries on Graphs with Data, in: ICDT, 2012, pp. 74–85.

[20] H. Sakamoto, D. Ikeda, Intractability of decision problems for finite-memory automata, Theor. Comput. Sci. 231 (2000) 297–308.

[21] T. Tan, Graph reachability and pebble automata over infinite alphabets, in: LICS, 2009, pp. 157–166.

[22] M. Kaminski, T. Tan, Tree automata over infinite alphabets, in: Pillars of Computer Science, 2008, pp. 386–423.

[23] A. Tal, Decidability of inclusion for unification based automata, Master's thesis, Department of Computer Science, Technion - Israel Institute of Technology, 1999.

[24] E. V. Kostylev, J. L. Reutter, D. Vrgoč, Containment of data graph queries, in: ICDT, 2014, pp. 131–142.

[25] M. Sipser, Introduction to the Theory of Computation, PWS Publishing, 1997.

[26] I. Glaister, J. Shallit, A lower bound technique for the size of nondeterministic finite automata, Information Processing Letters 59 (1996) 75–77.

[27] J. E. Hopcroft, R. Motwani, J. D. Ullman, Introduction to automata theory, languages, and computation - international edition (2. ed), Addison-Wesley, 2003.

[28] D. Vrgoč, Querying graphs with data, PhD thesis, The University of Edinburgh, 2014.