Journal of Computer and System Sciences ••• (••••) •••–•••

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Regular expressions for data words

Leonid Libkin [a], Tony Tan [b], Domagoj Vrgoč [a,c,*]

[a] University of Edinburgh, United Kingdom
[b] Hasselt University and Transnational University of Limburg, Belgium
[c] PUC Chile and Center for Semantic Web Research, Chile

ARTICLE INFO

Article history:
Received 2 May 2014
Received in revised form 27 October 2014
Accepted 3 March 2015
Available online xxxx

Keywords:
Data words
Register automata
Regular expressions

ABSTRACT

In this paper we define and study regular expressions for data words. We first define *regular expressions 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. We also study the complexity of the standard decision problems for REM, which turn 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 the applications of data words: *regular expressions with binding* (REWB) and *regular expressions with equality* (REWE). We study their expressive power and analyse the complexity of their standard decision problems. We also establish the following strict hierarchy of expressive power: REM is strictly stronger than REWB, and in turn REWB is strictly stronger than REWE.

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.

They were introduced by Kaminski and Francez in [14] who also proposed a natural extension of finite automata for them, called *register automata*. They 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, as well as in query evaluation tasks. For example, when reasoning about paths in XML trees, we may want to reason, not only about the labels of the trees, i.e. the XML tags, but also the values of attributes which can come from an infinite domain.

While the applications of logic and automata models for the structural part of XML (without data values) are well understood by now [17,21,25], taking into account the data values presents entirely new challenges [23,26]. Most efforts in this direction concentrate on finding "well behaved" logics and their associated automata [6,5,4,10], usually with the focus on finding logics with decidable satisfiability problem. In [23] Neven et al. studied the expressive power of various data word automata models in comparison with some fragments of FO and MSO. A well-known result of Bojanczyk et al. [4] shows that FO^2, the two-variable fragment of first-order logic extended by equality test for data values, is decidable over

* Corresponding author.
E-mail address: domagojvrgoc@gmail.com (D. Vrgoč).

http://dx.doi.org/10.1016/j.jcss.2015.03.005
0022-0000/© 2015 Elsevier Inc. All rights reserved.

data words. Recently, Ley et al. showed in [3,8] that the guarded fragment of MSO defines data word languages that are recognized by non-deterministic register automata.

Data words appear in the area of verification as well. In several applications, one would like to deal with concise and easy-to-understand representations of languages of data words. For example, in modelling infinite-state systems with finite control [9,12], a concise representation of system properties is much preferred to long and unintuitive specifications given by, say, automata.

The need for a good representation mechanism for data word languages is particularly apparent in graph databases [1] – a data model that is increasingly common in applications including social networks, biology, semantic Web, and RDF. Many properties of interest in such databases can be expressed by regular path queries [22], i.e. queries that ask for the existence of a path conforming to a given regular expression [7,2]. Typical queries are specified by the closure of atomic formulae of the form $x \xrightarrow{L} y$ under the "and" operation and the existential quantifier $\exists$. Intuitively, the atomic formula $x \xrightarrow{L} y$ asks for all pairs (x, y) of nodes such that there is a path from x to y whose label is in the regular language L [7].

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 [20]; 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* (RA), first introduced by Kaminski and Francez in [14] and studied, for example, in [9,24]. 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 [23,28], data automata [4], and class automata [5]. 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 [20] 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 expressions 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 by Kaminski and Tan in [15], but it fell short of even the first goal. In fact, the expressions of [15] 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 [9]. It is worthwhile noting that REM were also used in [20] and [16] to define a class of graph database queries called regular queries with memory (RQM).

However, one may argue that the binding rule in REM may not be intuitive. Consider the following expression: $a \downarrow x(a[x^{=}]a \downarrow x)^{*}a[x^{=}]$. Here the last comparison $x^{=}$ is not done with a value stored in the first binding, as one would expect, but with the value stored inside the scope of another binding (the one under the Kleene star). That is, the expression re-binds 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). We show that they are strictly weaker than REM. The nonemptiness problem for REWB drops to NP-complete, while the complexity of the membership problem remains the same as for REM.

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. Similar to REM, these expressions were used in [20,16] to define a class of graph queries called regular queries with data tests (RQD).

We would like to note that this paper combines and extends results from [19] and [18] where they appeared in a preliminary form and often without complete proofs. Here we present full proofs of all the results.

Organization 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 sequence of $\Sigma \times \mathcal{D}$, where Σ is a finite set of letters and $\mathcal{D}$ an infinite set of data values. That is, in data word each position carries a letter from Σ and a data value from $\mathcal{D}$. We will write data words as $\binom{a_1}{d_1}\ldots\binom{a_n}{d_n}$, 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 the 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 letter from the current position and comparing the data value to the ones previously stored in the registers. In this paper we use a slightly modified version of RA. The differences between this version and the other ones present in the literature [14,9,26] will be discussed in Section 3.

To test (in)equalities of data values we use the notion of *conditions* which are boolean combinations of atomic $=, \neq$ comparisons. Formally, conditions are defined as follows. Let $x_1, \ldots, x_k$ be variables, denoting the registers. The class of conditions $\mathcal{C}_k$ is defined by the grammar:

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

where $x_i \in \{x_1, \ldots, x_k\}$. 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 $x_1, \ldots, x_k$ is a partial function $\nu : \{x_1, \ldots, x_k\} \to \mathcal{D}$. We denote by $\mathcal{F}(x_1, \ldots, x_k)$ 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 $d, \nu \models \varphi_2$.

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:

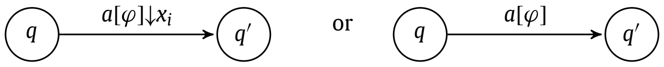
- Q is a finite set of states;
- $q_0 \in Q$ is the initial state;
- $F \subseteq Q$ is the set of final states;
- T is a finite set of transitions of the form:

$$q, a[\varphi]{\downarrow}x_i \to q' \quad \text{or} \quad q, a[\varphi] \to q',$$

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 $d, \nu \models \varphi$, where ν indicates 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 $\nu' = \nu$.

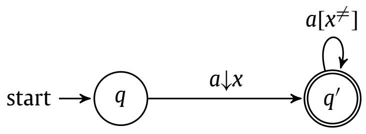
Let $w = \binom{a_1}{d_1} \ldots \binom{a_n}{d_n}$. A *run* of $\mathcal{A}$ on w is a sequence of configurations $(q_0, \nu_0), \ldots, (q_n, \nu_n)$ such that (q_0, ν_0) is the initial configuration and $(q_{i-1}, \nu_{i-1}) \vdash_{a_i,d_i} (q_i, \nu_i)$, for each $i = 1, \ldots, n$. It is called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

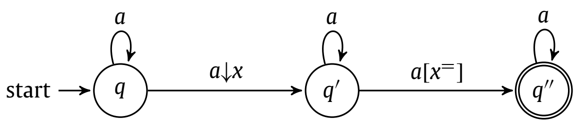
We now present a few examples of register automata.

Example 2.2. In the following let $\Sigma = \{a\}$. To ease the notation in transitions that have $\varphi = \texttt{tt}$ 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 some basic language theoretic properties of RA, which will be useful later on. Most of them already follow from [14]. However, we would like to note first that there are two versions of RA present in the literature. Although they are equivalent in expressive power, their difference subtly effects the complexity of decision problems discussed in this paper.

We will briefly discuss both versions, and show how they relate to the model presented in this paper

- In the first version different registers always contain different data values, as defined in [14]. In short, transitions in this version are of the form (q, i, q'), which intuitively means that the automaton can move from state q to q', if the data value currently read is the same as the one in register i.
- In the second version different registers may contain the same data value, as defined in [14,9,26]. In short, transitions in this version are of the form (q, I, q'), where I is a set of registers, which intuitively means that the automaton can move from state q to q', if the data value currently read can be found in registers in I.

For more details, we refer the reader to [14,9,26]. Let's call the former and the latter versions the S-version and the M-version, respectively. S stands for single register and M for multiple registers. It was shown in [14] that both versions have the same expressive power.

Note that transitions in S- and M-versions can be written in terms of conditions. For transitions in S-version, we have conditions of the form

$$x_i^= \wedge \bigwedge_{j \neq i} x_j^{\neq},$$

For transitions in M-version, we have conditions of the form

$$\bigwedge_{h \in I} x_h^= \wedge \bigwedge_{h \notin I} x_h^{\neq}.$$

Conversely, conditions can be easily translated to transitions in M-version. A condition is simply a class of subsets of registers that contain the currently read data value. Hence, all the models: the S-version, the M-version and the model presented in this paper are equivalent in expressive power.

The reader may ask though why we choose the model that allows repetition of data values. There are several reasons for this. First and foremost, RA defined in this paper provide greater flexibility in presenting our ideas than those studied so far, and thus, make the presentation less technical. Second, this model has by now became more mainstream when discussing register automata [9,26]. Finally, the original motivation for studying expressions over data words comes from the area of graph databases [20], where it is natural to assume that some data value can be repeated and reused afterwards for different purposes.

We will now list some general properties of register automata, which will be useful later on. We start with a folklore which states that RA can only keep track of as many data values as can be stored in their registers. Formally, we have:

Lemma 3.1. *(See [14, Proposition 4].) 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 the lemma in [14] assumes that in each assignment all the data values are different. It is straightforward to see that the proof also works when we allow repeated occurrence of data values.

Next, note that when restricted only to a finite set D of data values, register automata can be viewed as ordinary NFAs. To see this observe that there are only $|D|+1$ number of possible contents of each register: either the register is empty or it contains a data value from D. Hence, all possible contents of the registers can be simply encoded inside the states of the NFA. We formally state this in the lemma below.

Lemma 3.2. *(See [14, 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 and nonemptiness problems are some of the most important decision problems related to formal languages. To be precise, we present the definitions of these problems here. The *nonemptiness problem* asks us to check, given as input an RA $\mathcal{A}$, whether $L(\mathcal{A}) \neq \emptyset$. The *membership problem* takes as input an RA $\mathcal{A}$ and a data word w, and requires to check whether $w \in L(\mathcal{A})$.

We now recall the exact complexity of these problems for register automata.

Fact 3.3.

- The nonemptiness problem for RA is PSPACE-complete *[9]*.
- The membership problem for RA is NP-complete *[24]*.

The hardness in [9] is for the M-version, while the one in [24] is for the S-version. However, note that translating the transitions in S- and M-versions to conditions incurs only linear blow-up (for S-version) and no blow-up (for M-version). Hence, the NP-hardness and the PSPACE-hardness for decision problems for S- and M-versions will also hold for the RA defined in this paper. The upper bound follows from Lemmas 3.1 and 3.2 and Fact 3.4.

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 mentioned in [14], of which the proofs can be easily modified to suit the RA model proposed here.

Fact 3.4. (See [14].)

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 $\mathcal{A}$.

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 $\Sigma[x_1, \dots x_k]$ and $\nu, \nu' \in \mathcal{F}(x_1, \dots, x_k)$. 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:

$$e_1 = (a{\downarrow}x) \cdot (a[x^{\neq}])^*$$

$$e_2 = a^* \cdot (a{\downarrow}x) \cdot a^* \cdot (a[x^{=}]) \cdot a^*$$

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

Example 4.3. Let $\Sigma = \{a, b_1, b_2, \dots, b_l\}$. 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 $(a[x^{=}] + b_1[x^{=}] + \dots + b_l[x^{=}])$. 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.

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

- *For every REM e over $\Sigma[x_1, \dots, x_k]$, 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 $\Sigma[x_1, \dots, x_k]$ such that $L(e_{\mathcal{A}}) = L(\mathcal{A})$. Moreover, the REM $e_{\mathcal{A}}$ can be constructed in exponential time.*

Proof. In what follows we will need the following notation. For an REM e over $\Sigma[x_1, \dots, x_k]$ and $\nu, \nu' \in \mathcal{F}(x_1, \dots, x_k)$, 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 $(q_0, \nu_0), \dots, (q_n, \nu_n)$ 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 4.5.

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 4.5. The structure of the proof follows the standard NFA-regular expressions equivalence, cf. [27], 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.

- 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 $T = \{q_0, a[\varphi] \to q_1\}$.
- The case when $e = e_1 + e_2$, or $e = e_1 \cdot e_2$, or $e = e_1^*$ follow from the inductive hypothesis and the fact that RA languages are closed under union, concatenation and Kleene star (Fact 3.4).

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 labels $a[\varphi]{\downarrow}x_i$, or $a[\varphi]$, but also by any regular expression with memory. The transition relation is called δ and 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.

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 REM. 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 is a standard procedure used to show the equivalence between NFAs and regular expressions (see [27]). For completeness, we repeat it here and show how it can also be used when data values are taken into account.

1. CONVERT($\mathcal{A}$)
2. Let n be the number of states of $\mathcal{A}$.
3. If $n = 2$, then $\mathcal{A}$ 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.
4. If $n > 2$, select any state q_{rip}, different from q_{start} and q_{end} and modify $\mathcal{A}$ in the following manner to obtain $\mathcal{A}'$ 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.
5. Return CONVERT($\mathcal{A}'$).

We now prove that for any GRA $\mathcal{A}$, the expression CONVERT($\mathcal{A}$) and GRA $\mathcal{A}$ recognize the same language of data words. We do so by induction on the number n of states of our GRA $\mathcal{A}$. If $n = 2$ then $\mathcal{A}$ 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 $\mathcal{A}$.

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

To see this assume first that $w \in L(\mathcal{A}, \sigma, \sigma')$, i.e. $\mathcal{A}$ 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 $\mathcal{A}'$, so $w \in L(\mathcal{A}', \sigma, \sigma')$, since all the arrows present here are also in $\mathcal{A}'$.

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 $\mathcal{A}'$, 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 $((R_1)(R_2)^*(R_3), w_i w_{i+1} \ldots w_j, \sigma) \vdash \sigma'$, so $\mathcal{A}'$ can jump from c_i to c_j using only one transition.

Conversely, suppose that $w \in L(\mathcal{A}', \sigma, \sigma')$. This means that there is a computation of $\mathcal{A}'$, starting with the initial content of registers σ and ending with σ'. We know that each arrow in $\mathcal{A}'$ from q_i to q_j goes either directly (in which case it is already in $\mathcal{A}$) or through q_{rip} (in which case we use the definition of acceptance by regular expressions to unravel this word into part recognized by $\mathcal{A}$). In either case we get an accepting run of $\mathcal{A}$ 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. This completes our proof of Theorem 4.4. □

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.

The next theorem states that the nonemptiness and the membership problems for REM are PSpace-complete and NP-complete, respectively. Note that the hardness results do not follow from Theorems 3.3 and 4.4, since the conversion from RA to REM in Theorem 4.4 takes exponential time.

Theorem 4.6.

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

Proof. We begin by proving the bound for nonemptiness. By Theorem 4.4, 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.

Next we are going to establish the PSpace-hardness, which is established via a reduction from the nonuniversality problem of finite state automata. The nonuniversality problem asks, given a finite state automaton $\mathcal{A}$ as input, 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 $\overline{\mathcal{A}}$.

That is, we are trying to simulate a sequence $S_0, S_1, \ldots, S_m$ of subsets of Q, where:

(P1) $S_0 = \{q_1\}$ and $S_m \cap F = \emptyset$;
(P2) for each $i = 1, \ldots, m$, there is $b \in \Sigma$ such that $S_i = \{q \mid \exists p \in S_{i-1}$ such that $q \in \delta(p, b)\}$.

To do so we designate two distinct data values, t and f, and encode each subset $S \subseteq Q$ 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 the subset S. Since we are checking reachability, we will need only to remember the current set S_j and the next set S_{j+1}. In what follows we will encode 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 encode data words that describe a sequence $S_0, \ldots, S_m$ that satisfies (P1) and (P2) above by demonstrating how one can move from one set S_j to the next set S_{j+1} (as witnessed by their codes in current state tape and next state tape), starting with the initial set $S_0 = \{q_1\}$ and ending in set S_m, where $S_m \cap F = \emptyset$.

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 to variables named t and f respectively.

The first expression we need is:

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

This expression encodes two different values as t and f and initializes current state tape to contain the 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:

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

This expression is used to check that we have reached a state not containing any final state from F.

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:

$$\mathtt{update} := \bigvee_{b \in \Sigma} u_{\delta(q_1,b)} \cdot u_{\delta(q_2,b)} \cdots u_{\delta(q_n,b)}.$$

Here we use $\bigvee$ for union. 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:

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

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

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

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 sequence $S_0, \ldots, S_m$ that satisfies (P1) and (P2) above. That is, there is a word w from Σ^* not in the language of $\mathcal{A}$. This sequence 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 $s_1, \ldots, s_n$) to initial set $S_0 = \{q_1\}$, then runs the updates of the tape according to w and finally ends in a set S_m, where $S_m \cap F = \emptyset$.

Conversely, each word in $L(e)$ corresponds to a sequence $S_0, \ldots, S_m$ that satisfies (P1) and (P2) above. That is, the part of word corresponding to init sets $S_0 = \{q_1\}$. Then the part of this word that corresponds to step* corresponds to updating our tapes in a way that properly encodes one step from S_i to S_{i+1}. Finally, end denotes that we have reached a set S_m where $S_m \cap F = \emptyset$.

Next we establish the NP bound for the membership problem. By Theorem 4.4, 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.

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 a 3-CNF formula, where each $\ell_{i,j}$ is a literal. We will construct a data word w and a regular expression with memory e over a single letter alphabet $\Sigma = \{a\}$, both of length linear in the length of φ, such that φ is satisfiable if and only if $w \in L(e)$.

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

$$w = \left(\binom{a}{0}\binom{a}{1}\right)^k \left(\binom{a}{d_{\ell_{1,1}}}\binom{a}{d_{\ell_{1,2}}}\binom{a}{d_{\ell_{1,3}}}\right) \cdots \left(\binom{a}{d_{\ell_{n,1}}}\binom{a}{d_{\ell_{n,2}}}\binom{a}{d_{\ell_{n,3}}}\right),$$

where $d_{\ell_{i,j}} = 1$, if $\ell_{i,j} = x_m$, for some $m \in \{1, \ldots k\}$ and 0, if $\ell_{i,j} = \neg x_m$.

The idea behind this data word is that with the first part that corresponds to the variables, i.e. with $(\binom{a}{0}\binom{a}{1})^k$, 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:

$$e = (a^* \cdot a{\downarrow}x_1) \cdot (a^* \cdot a{\downarrow}x_2) \cdot (a^* \cdot a{\downarrow}x_3) \cdots (a^* \cdot a{\downarrow}x_k) \cdot a^* \cdot \text{clause}_1 \cdot \text{clause}_2 \ldots \text{clause}_n,$$

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

$$\text{clause}_i = a[x^{=}_{j_1}] \cdot a \cdot a + a \cdot a[x^{=}_{j_2}] \cdot a + a \cdot a \cdot a[x^{=}_{j_3}].$$

We now prove that φ is satisfiable if and only if $w \in L(e)$. For the only if direction, assume 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.

For the converse, assume 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, called *regular expressions with binding* (REWB) whose variables adhere to the usual scoping rules familiar from programming languages or first order logic.

As mentioned in the introduction, REWBs have a similar syntax, but rather different semantics than REM. REMs 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 4.4 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 than REMs. Therefore, proper binding of variables comes with a cost – albeit small – in terms of expressiveness.

As before, $\mathcal{C}_k$ denotes the class of conditions over the variables $x_1, \ldots, x_k$.

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:

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

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. In what follows we will also use the abbreviation a in place of $a[\mathtt{tt}]$.

Semantics Let $r(\bar{x})$ 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 $\nu(\bar{x})$ is defined.

A regular expression $r(\bar{x})$ 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 $r = a[\varphi]$, 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 $L(r, \nu) = \bigcup_{d \in \mathcal{D}} \left\{\begin{pmatrix} a \\ d \end{pmatrix}\right\} \cdot L(r_1, \nu[x_i \leftarrow d])$.

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

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

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 [14].

The theorem below states that REWB languages are not closed under intersection. Since the proof is rather lengthy and uses several auxiliary lemmas we defer it to Section 5.1.

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

Since REWB are subsumed by RA and the latter are closed under intersection we immediately obtain the following.

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

Note that as a separating example we could take the language $\mathcal{L}_1 \cap \mathcal{L}_2$ defined in Section 5.1. However, this 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 behavior of REWB.

Theorem 5.5.

- *The nonemptiness problem for REWB is* NP*-complete.*
- *The membership problem for REWB is* NP*-complete.*

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

Proposition 5.6. *For every REWB r over $\Sigma[x_1, \dots, x_k]$ 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 at most $|r|$.*

Proof. The proof is by induction on the length of r. The basis is when the length of r is 1. In this case, r is $a[\varphi]$ 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 $|w_1| \leq |r_1|$; or
  - there exists $w_2 \in L(r_2, \nu)$ such that $|w_2| \leq |r_2|$.

  Thus, by definition, there exists $w \in L(r, \nu)$ such that $|w| \leq |r|$.
- 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 $|w_1| \leq |r_1|$; and
  - there exists $w_2 \in L(r_2, \nu)$ such that $|w_2| \leq |r_2|$.

  Thus, by definition, $w_1 \cdot w_2 \in L(r, \nu)$ and $|w_1 \cdot w_2| \leq |r|$.
- 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 $|w_1| \leq |r_1|$. By definition, $\binom{a}{d} w_1 \in L(r, \nu)$.

This completes the proof of Proposition 5.6. □

The NP membership now follows from Proposition 5.6, where given an 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 (Theorem 5.5, second item). Since the two NP algorithms are independent they can be composed.

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 $x_1, \dots x_k$ 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 $\Sigma[y_1, z_1, y_2, z_2, \dots, y_k, z_k]$, with $\Sigma = \{a\}$, of length $\mathcal{O}(n)$ such that φ is satisfiable if and only if $L(r) \neq \emptyset$.

Let r be the following REWB.

$$r := a \downarrow_{y_1} .\{a \downarrow_{z_1} .\{a \downarrow_{y_2} .\{a \downarrow_{z_2} .\{\cdots \{a \downarrow_{y_k} .\{a \downarrow_{z_k} .\{$$
$$(r_{1,1} + r_{1,2} + r_{1,3}) \cdots (r_{n,1} + r_{n,2} + r_{n,3})\}\} \dots\},$$
$$r_{i,j} := \begin{cases} a[y_k^= \wedge z_k^=] & \text{if } \ell_{i,j} = x_k \\ a[y_k^= \wedge z_k^{\neq}] + a[z_k^= \wedge y_k^{\neq}] & \text{if } \ell_{i,j} = \neg x_k \end{cases}$$

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 $f : \{x_1, \dots, x_k\} \mapsto \{0, 1\}$ making φ true. We define the evaluation $\nu : \{y_1, z_1, \dots y_n, z_n\} \mapsto \{0, 1\}$ 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.

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

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

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

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

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

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

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

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

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

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

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

- If $r_{j,1} = a[y_i^= \wedge z_i^=]$, 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 $r_{j,1} = a[y_i^= \wedge z_i^{\neq}] + a[z_i^= \wedge y_i^{\neq}]$, 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 of the nonemptiness problem.

Next we show that membership problem is also NP-complete.

The upper bound follows from Theorem 3.3 and the fact that every REWB can be translated into an equivalent RA in polynomial time (using the same transition as in the proof of Theorem 4.4).

For the lower bound we observe that in the proof of NP lower bound in Theorem 4.6 we can use the following REWB:

$$e = a^* \cdot a \downarrow_{x_1} .\{a^* \cdot a \downarrow_{x_2} .\{a^* \cdot a \downarrow_{x_3} .\{\cdots a^* \cdot a \downarrow_{x_k} . \{a^* \cdot \text{clause}_1 \cdot \text{clause}_2 \ldots \text{clause}_n\}\} \cdots \},$$

while all the other details of the proof remain the same. □

Recall that 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.

5.1. Proof of Theorem 5.3

Let $\mathcal{L}_1$ 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 $d_2 = d_5, d_6 = d_9, \ldots, d_{4n-6} = d_{4n-3}$.

Similarly, we define $\mathcal{L}_2$ be the language of words of the above form where $d_4 = d_7, d_8 = d_{11}, \ldots, d_{4n-4} = d_{4n-1}$.

In particular, $\mathcal{L}_1 \cap \mathcal{L}_2$[1] is the language consisting 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 $\mathcal{L}_1$ and $\mathcal{L}_2$ are REWB languages. In the following we are going to show the following.

Lemma 5.7. *$\mathcal{L}_1 \cap \mathcal{L}_2$ is not an 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 a subexpression of r and $\nu \in \mathcal{F}(x_1, \ldots, x_k)$, constructed as follows. The root node is labeled with $(e, \emptyset)$. If a node u is labeled with (r', ν), then its children are labeled as specified below.

- If $r' = a[\varphi]$, then u has exactly one child: a leaf labeled with $\binom{a}{d}$ such that $d, \nu \models \varphi$.
- If $r' = r_1 + r_2$, then u has exactly one child: a node labeled with either (r_1, ν) or (r_2, ν).
- If $r' = r_1 \cdot r_2$, then u has exactly 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 at least one child: either a leaf labeled with ϵ; or several nodes labeled with (r_1, ν).
- If $r' = a \downarrow_x .\{r_1\}$, then u has exactly 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 leaves of t from left to right.

Proposition 5.8. *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.

[1] Note that this is the same language used in [14], Example 3, to show that the pumping lemma does not apply to RA.

- 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.

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

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 5.9. *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 $\mathcal{L}_1 \cap \mathcal{L}_2$ 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 $L(r) = \mathcal{L}_1 \cap \mathcal{L}_2$, 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.

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

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

Claim 5.10. *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 $w'' \notin \mathcal{L}_1 \cap \mathcal{L}_2$, hence, contradicts the fact that $L(r) = \mathcal{L}_1 \cap \mathcal{L}_2$. 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 bound only once. Otherwise, we can rename the variable.

Claim 5.11. *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 same 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 5.12 below immediately implies that Lemma 5.7.

Claim 5.12. *There exists a word $w'' \notin \mathcal{L}_1 \cap \mathcal{L}_2$, such that $w'' \in L(r)$.*

Proof. The word w'' is constructed from the word w. By Claim 5.11, 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 $\mathcal{L}_1 \cap \mathcal{L}_2$, in 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 $h = \max\{l \in B : h_1 < l < h'_1\}$. The index h is not the index of the last e, hence $R_t(h)$ exists and $R_t(h) < h_2$, by Proposition 5.9. 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 $\mathcal{L}_1 \cap \mathcal{L}_2$, by definition. Thus, this completes the proof of our claim. □

This completes our proof of Lemma 5.7. Since both $\mathcal{L}_1$ and $\mathcal{L}_2$ are easily definable by an REWB using only one variable, this completes the proof of Theorem 5.3.

6. Regular expressions with equality

In this section we define yet another kind of expressions, regular expressions with equality (REWE), 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.

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

Proof. 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*, and show that they subsume regular expressions with equality and that they cannot 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 is a condition and $I \subseteq \{x_1, \dots, x_k\}$ 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 and not in I 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 cannot 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.

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 the following must occur: There is a data value d_i, with $i \geq 2$, such that when the automaton reads 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. We can then replace d_i with d_1 and get an accepting run. Note that this happens because the accepting run can only test for inequalities and d_1 is by assumption different from all the other values appearing in the word. Thus we obtained an accepting run on a word that has the first data value repeated – 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 6.4. *For every REWE 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 of the lemma is almost identical to the proof of item one in Theorem 4.4. We can view this as introducing a new variable for every $=, \neq$ comparison in e and act as the subexpression $f_=$ reads $(a{\downarrow}x) \cdot e' \cdot (b[x^=])$, where $f = a \cdot e' \cdot b$ and analogously for $\neq$. Here we use the same conventions as in the proof of item one of the proposition and assume that all expressions are well-formed. Note that in this case all variables come with their scope, so we do not have to worry about transferring register configurations from one side of the construction to another (for example when we do concatenation). The underlying automata remain the same. □

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 6.5. *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 [14]. 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 observe that $\mathcal{L}_1$ and $\mathcal{L}_2$ from the proof of Theorem 5.3 are clearly recognizable by REWE. This and Theorem 6.3 imply the desired result. □

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 REWE 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 6.6. *Let D be a finite set of data values. For any REWE 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 $\mathcal{O}(|e|)$ states and $\mathcal{O}(|e| \times (|D|^2 + |e|))$ transitions, and can be constructed in polynomial time.*

Proof. When talking about PDA we will use the notational conventions from [13]. We will assume that we do not use expressions $e = \varepsilon$ and $e = \emptyset$ to avoid some technical issues. 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.

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 now describe how to construct, by induction on e, a nondeterministic PDA $P_{e,D}$ over the alphabet $\Sigma \times D$ such that:

- $w = \binom{a_1}{d_1} \dots \binom{a_n}{d_n}$ is accepted by $P_{e,D}$ if and only if $w \in L(e)$ and $d_1, \dots, d_n \in D$.
- 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. We also note that our PDA will use a bounded stack (no larger than the size of the expression itself) that is written from bottom to top.

The construction is inductive on the structure of the expression e. The case when $e = a$ is straightforward, and for $e = e_1 + e_2, e_1 \cdot e_2$ and e_1^+ we use standard constructions [13].

The only interesting case is when $e = (e_1)_=$ and $e = (e_1)_{\neq}$. The automaton $P_{e,D}$ in both cases is constructed such that it begins by pushing the first data value of the input word onto the stack. It then marks this data value with a new stack symbol X_e (i.e. it pushes a new stack symbol used to denote this (in)equality test onto the stack) and runs the automaton for $P_{e_1,D}$ which exists by the induction hypothesis. Every time we enter a final state of $P_{e_1,D}$ our automaton $P_{e,D}$ will empty the stack until it reaches the first data value (here we use the new symbol X_e) and check if it is equal or different to the last data value of the input word.

The extra technical conditions are used to ensure that the automaton works correctly. It is straightforward to see that the automaton is linear in the length of the expression and uses at most polynomially many transitions. □

Using this result we can show the PTIME bound.

Theorem 6.7. *The nonemptiness problem and the membership problem for REWE are in* PTIME.

Proof. For nonemptiness, let e be the given REWE. Since REWE is a special form of REWB, by Proposition 5.6, if $L(e) \neq \emptyset$, then there exists a data word w in which there are at most $|e|$ different data values. We can then construct the PDA $P_{e,D}$, where $D = \{0, 1, \dots, |e| + 1\}$ in time polynomial in $|e|$. Since the nonemptiness of PDA can be checked in PTIME, we obtain the PTIME upper bound for 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 $D = \{0, 1, \dots, n\}$, 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 we demonstrate below.

Corollary 6.8. *For every REWE 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. Note that in the proof of Lemma 6.6 all of the PDA use bounded stack. If the stack in a PDA has only up to N symbols, we can simulate the PDA with NFA by encoding all possible contents of the stack in the states of the NFA. Such simulation will incur an exponential blow-up. Suppose M is the number of states in the PDA and Γ is the set of stack symbols, the total number of states in the NFA will be $\mathcal{O}(\mathcal{M}|\Gamma|^N)$. This immediately implies the desired result. □

However, the exponential lower bound is the best we can do in the general case. Namely, we have the following proposition.

Proposition 6.9. *There is a sequence of regular expressions with equality $\{e_n\}_{n\in\mathbb{N}}$, over the alphabet $\Sigma = \{a\}$, each of length linear in n, such 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 size exponential in $|e_n|$.*

Proof. To prove this we will use the following theorem for proving lower bounds of NFAs [11]. Let $L \subseteq \Sigma^*$ be a regular language and suppose there exists a set $P = \{(x_i, y_i) : 1 \le i \le n\}$ of pairs such that:

1. $x_i \cdot y_i \in L$, for every $i = 1, \dots n$, and
2. $x_i \cdot y_j \notin L$, for $1 \le i, j \le n$ 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 $w_1, \dots w_{2^n}$ be a list of all the words 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 [11], that any NFA recognizing $L(e_n)$ has at least $\mathcal{O}(2^{|e_n|})$ states. □

Note that this lower bound is essentially the same as the one for translating PDA with bounded stack in NFA.

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. 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 [20], or modelling infinite-state systems with finite control [9].

For the sake of completeness, we also include results on another model of automata for data words called *variable automata*, or VFA for short, where variables are used to store data values. Originally introduced in [12] to reason about languages over infinite alphabets, they can easily be extended to operate over data words [29]. They define a class of

Table 1
Closure properties of data word defining formalisms.

	RA	REM	REWB	REWE	VFA[*]
Union	+	+	+	+	+
Intersection	+	+	−	−	+
Concatenation	+	+	+	+	+
Kleene star	+	+	+	+	+
Complement	−	−	−	−	−

[*] Results on VFA follow from [12].

Table 2
Complexity of main decision problems.

	RA	REM	REWB	REWE	VFA[*]
Nonemptiness	PSPACE-c	PSPACE-c	NP-c	PTIME	NLOGSPACE-c
Membership	NP-c	NP-c	NP-c	PTIME	NP-c
Universality	undecidable	undecidable	undecidable	undecidable[†]	undecidable

[*] Results on VFA are from [12].
[†] was first shown in [16].

languages orthogonal to the ones captured by formalisms in this paper. All of the complexity bounds and closure properties of VFA were already established in [12] and they readily extend to the setting of data words.

Table 1 presents the basic closure properties of the languages studied in this paper, as well as VFA languages. 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 show that this is no longer true.

The second class of problems we studied was the complexity of nonemptiness and membership. This is summarized in Table 2. We showed that with the increase of expressive power of a language the complexity of the two problems rises correspondingly. 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 complexity for membership than for nonemptiness. Note that the bounds we established for REWE are not tight and it would be interesting to see if the complexity of membership and nonemptiness can be brought down even further in this case.

Table 2 also include *universality problem*, which asks, given as input an automaton(or an expression) $\mathcal{A}$, to check whether $L(\mathcal{A}) = (\Sigma \times \mathcal{D})^*$. While the undecidability of universality for RA was already established in [14], in [18] we showed that the problem remains undecidable even for REWBs. Recently, this result was strengthened by Kostylev et al. in [16] by showing that this is true even for REWE. For completeness we include these results in Table 2 that summarizes the complexity bounds for the problems considered in this paper.

Lastly, we also studied how the five classes of languages compare to one another. While REM were originally introduced as an expression analogue of RA, they subsume REWB and REWE. 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 [14]; while at the same time they cannot capture the language defined by the REWE $((aa)_=)^+$ (see [12] for details). We thus obtain:

Corollary 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.*

Acknowledgments

The authors would like to thank the reviewers for their comments. This work was partially supported by the EPSRC Grants G049165 and J015377. Tan is also supported by FWO Pegasus Marie Curie fellowship and Vrgoč by the Millennium Nucleus Center for Semantic Web Research Grant NC120004.

References

[1] R. Angles, C. Gutierrez, Survey of graph database models, ACM Comput. Surv. 40 (1) (2008).
[2] 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.

[3] M. Benedikt, C. Ley, G. Puppis, Automata vs. logics on data words, in: CSL, 2010, pp. 110–124.
[4] M. Bojanczyk, C. David, A. Muscholl, T. Schwentick, L. Segoufin, Two-variable logic on words with data, ACM Trans. Comput. Log. 12 (4) (2011).
[5] 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.
[6] M. Bojanczyk, P. Parys, XPath evaluation in linear time, J. ACM 58 (4) (2011).
[7] D. Calvanese, G. De Giacomo, M. Lenzerini, M. Vardi, Rewriting of regular expressions and regular path queries, J. Comput. Syst. Sci. 64 (3) (2002) 443–465.
[8] T. Colcombet, C. Ley, G. Puppis, On the use of guards for logics with data, in: MFCS, 2011, pp. 243–255.
[9] S. Demri, R. Lazić, LTL with the freeze quantifier and register automata, ACM Trans. Comput. Log. 10 (3) (2009).
[10] D. Figueira, Satisfiability of downward xpath with data equality tests, in: 28th ACM Symposium on Principles of Database Systems, PODS, 2009, pp. 197–206.
[11] I. Glaister, J. Shallit, A lower bound technique for the size of nondeterministic finite automata, Inf. Process. Lett. 59 (2) (1996) 75–77.
[12] 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.
[13] J.E. Hopcroft, R. Motwani, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, international edition (2nd ed.), Addison–Wesley, 2003.
[14] M. Kaminski, N. Francez, Finite memory automata, Theor. Comput. Sci. 134 (2) (1994) 329–363.
[15] M. Kaminski, T. Tan, Tree automata over infinite alphabets, in: Pillars of Computer Science, 2008, pp. 386–423.
[16] E.V. Kostylev, J.L. Reutter, D. Vrgoč, Containment of data graph queries, in: ICDT, 2014, pp. 131–142.
[17] L. Libkin, Logics for unranked trees: an overview, Log. Methods Comput. Sci. 2 (3) (2006).
[18] L. Libkin, T. Tan, D. Vrgoč, Regular expressions with binding over data words for querying graph databases, in: DLT, 2013.
[19] L. Libkin, D. Vrgoč, Regular expressions for data words, in: LPAR, 2012, pp. 274–288.
[20] L. Libkin, D. Vrgoč, Regular path queries on graphs with data, in: ICDT, 2012, pp. 74–85.
[21] M. Marx, Conditional xpath, ACM Trans. Database Syst. 30 (4) (2005) 929–959.
[22] A. Mendelzon, P. Wood, Finding regular simple paths in graph databases, SIAM J. Comput. 24 (6) (1995) 1235–1258.
[23] F. Neven, T. Schwentick, V. Vianu, Finite state machines for strings over infinite alphabets, ACM Trans. Comput. Log. 5 (3) (2004) 403–435.
[24] H. Sakamoto, D. Ikeda, Intractability of decision problems for finite-memory automata, Theor. Comput. Sci. 231 (2) (2000) 297–308.
[25] T. Schwentick, Automata for XML – a survey, J. Comput. Syst. Sci. 73 (3) (2007) 289–315.
[26] L. Segoufin, Automata and logics for words and trees over an infinite alphabet, in: CSL, 2006, pp. 41–57.
[27] M. Sipser, Introduction to the Theory of Computation, PWS Publishing, 1997.
[28] T. Tan, Graph reachability and pebble automata over infinite alphabets, in: LICS, 2009, pp. 157–166.
[29] D. Vrgoč, Querying graphs with data, PhD thesis, The University of Edinburgh, 2014.