Regular Expressions for Data Words

Regular Expressions for Data Words

Leonid Libkin and Domagoj Vrgoč

School of Informatics, University of Edinburgh

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

Our goal, therefore, is to define and study regular expressions for data words. As the automaton model, we take register automata, which are a natural analog of NFAs for data words. We first equip standard regular expressions with limited memory, and show that they capture the class of data words defined by register automata. The complexity of the main decision problems for these expressions (nonemptiness, membership) also turns out to be the same as for register automata. We then look at a subclass of these regular expressions that can define many properties of interest in applications of data words, and show that the main decision problems can be solved efficiently for it.

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 an alphabet $\Sigma = \{a, b\}$ and $\mathbb{N}$ as the domain of values. It can be viewed as the ordinary word abb in which the first and the third positions are equipped with value 1, and the second position with value 2.

These were introduced in [13] which proposed a natural extension of finite automata for them, called *register automata*. Data words have become an active subject of research lately due to their applications in XML, in particular in static analysis of logic and automata-based XML specifications, and in query evaluation tasks. Indeed, paths in XML trees should account not only for the labels (XML tags) but values of attributes, which can come from an infinite domain, such as $\mathbb{N}$. While logic and automata models are well-understood by now for the structural part of XML (i.e., trees) [15, 17, 22], adding data values required a concentrated effort for finding good logics and their associated automata [4, 6, 5, 10, 20, 23]. Connections between logical and automata formalisms have been explored as well, usually with the focus on finding logics with decidable satisfiability problem. A well-known result of [5] shows that FO^2, the two-variable fragment of first-order logic extended by equality test for data values, is decidable over data words. Another account of this was given in [20], where various data word automata models are compared to fragments of FO and MSO with regard

to their expressive power. Recently, the problem was studied in [3, 8]; in particular it was shown that the guarded fragment of MSO defines data word languages that are recognized by non-deterministic register automata.

Data words appear in other areas as well, in particular verification, and querying databases. In several applications, one would like to deal with concise and easy-to-understand representations of languages of data words. These can be used, for example, in extending languages for XML navigation that take into account data values. Another possible example is in the field of verification, in particular from modeling infinite-state systems with finite control [9, 12]. Here having a concise representation of system properties is much preferred to long and unintuitive specifications given by e.g. automata.

The need for a good representation mechanism for data word languages is particularly apparent in the area of querying graph databases [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 are expressed by regular path queries [18], asking for the existence of a path conforming to a given regular expression, or their extensions [7, 2]. Typical queries are specified by the closure of atomic formulae $x \xrightarrow{L} y$ under $\wedge$ and $\exists$; the atoms ask for the existence of a path whose label is in a regular language L between x and y [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 [16]; 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 rather cumbersome to be used in a query language).

The most natural extension of the usual NFAs to data words is *register automata*, first introduced in [13] and studied, for example, in [9, 21]. 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 [20, 25], data automata [5], and class automata [6]. In this paper we concentrate on languages specified by register automata, since they are the most natural generalization of finite state automata to languages over data words.

As mentioned earlier, if we think of a specification of a data word language, register automata are not the most natural way of providing them: in fact, even over the usual words, regular languages are easier to describe by regular expressions than by NFAs. For example, in XML and graph database applications, specifying paths via regular expressions is completely standard. In many XML specifications (e.g., XPath), data value comparisons are fairly limited: for instance, one checks if two paths ends 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 [16] 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.

Note that an attempt to find such regular expressions has been made in [14], but it fell short of even the first goal. In fact, the expressions of [14] 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.

We shall call such expressions *regular expressions with memory*. We formally define their semantics, give examples, prove that they capture register automata and share their algorithmic properties. We then introduce a different kind of regular expressions, *regular expressions with equality*. The previous language, for example, will be captured by the expression $(aa)_{\neq}$, saying that the finite part of the data word reads aa, and the data values at the beginning and at the end are different. We show that such expressions are strictly weaker than expressions with memory, but enjoy nice algorithmic properties.

Organization. In Section 2 we define register automata, and list their closure properties and complexity results about nonemptiness and membership. In Section 3 we introduce regular expressions with memory and show that they define the same class of languages as register automata. In Section 4 we introduce regular expressions with equality, show that while they are strictly weaker than register automata, they admit faster algorithms for decision problems that are based on the close connection of these expressions with pushdown automata. Due to space limitations, some proofs are only sketched, and complete proofs will appear in the full version of the paper.

2 Register automata over data words

A **data word** is simply a finite string over the alphabet $\Sigma \times \mathcal{D}$, where Σ is a finite set of letters and $\mathcal{D}$ an infinite set of data values. That is, in each position a data word carries a letter from Σ and a data value from $\mathcal{D}$. We will denote data words by $\binom{a_1}{d_1} \ldots \binom{a_n}{d_n}$, where $a_i \in \Sigma$ and $d_i \in \mathcal{D}$. The set of all data words over the alphabet Σ and set of data values $\mathcal{D}$ is denoted by $\Sigma[\mathcal{D}]^*$. A data word language is simply a subset $L \subseteq \Sigma[\mathcal{D}]^*$.

Register automata are an analog of NFAs for data words. They move from one state to another by reading the appropriate letter from the finite alphabet and comparing the data value to ones previously stored into the registers. Our version of register automata will use comparisons which are boolean combinations of atomic $=, \neq$ comparisons of data values.

To define such conditions formally, assume that, for each $k > 0$, we have variables $x_1, \ldots, x_k$. Then the set of conditions $\mathcal{C}_k$ is given by the grammar:

$$c \;:=\; \mathtt{tt} \mid \mathtt{ff} \mid x_i^{=} \mid x_i^{\neq} \mid c \wedge c \mid c \vee c \mid \neg c, \quad 1 \leq i \leq k.$$

The satisfaction is defined with respect to a data value $d \in \mathcal{D}$ and a tuple $\tau = (d_1, \ldots, d_k) \in \mathcal{D}^k$ as follows:

- $d, \tau \models \mathtt{tt}$ and $d, \tau \not\models \mathtt{ff}$;
- $d, \tau \models x_i^{=}$ iff $d = d_i$;
- $d, \tau \models x_i^{\neq}$ iff $d \neq d_i$;
- $d, \tau \models c_1 \wedge c_2$ iff $d, \tau \models c_1$ and $d, \tau \models c_2$ (and likewise for $c_1 \vee c_2$);
- $d, \tau \models \neg c$ iff $d, \tau \not\models c$.

In what follows, $[k]$ is a shorthand for $\{1, \ldots, k\}$.

Definition 1 (Register data word automata). *Let Σ be a finite alphabet and k a natural number. A k-*register data word automaton *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 $(q, a, c) \to (I, q')$, where q, q' are states, a is a label, $I \subseteq [k]$, and c is a condition in $\mathcal{C}_k$.*

Intuitively the automaton traverses a data word from left to right, starting in q_0, with all registers empty. If it reads $\binom{a}{d}$ in state q with register configuration τ, it may apply a transition $(q, a, c) \to (I, q')$ if $d, \tau \models c$; it then enters state q' and changes contents of registers i, with $i \in I$, to d.

To define acceptance formally we first define a configuration of a k-register data word automaton $\mathcal{A}$ on data word $w = \binom{a_1}{d_1} \ldots \binom{a_n}{d_n}$ as a triple (q, j, τ), where q is the current state of $\mathcal{A}$, j is the current position of the symbol in w that $\mathcal{A}$ reads and τ is the current state of the registers. We use the symbol $\perp$ to indicate that a register is unassigned; that is, τ is a k-tuple over $\mathcal{D}_\perp = \mathcal{D} \cup \{\perp\}$. The initial configuration is $(q_0, 1, \tau_0)$, where $\tau_0 = (\perp, \ldots, \perp)$, and any configuration (q, j, τ) with $q \in F$ is a final configuration.

From a configuration (q, j, τ) we can move to a configuration $(q', j + 1, \tau')$ if:

- $(q, a_j, c) \to (I, q')$ is a transition in $\mathcal{A}$,
- $d_j, \tau \models c$ and
- τ' is obtained from τ by replacing data values in registers from I by d_j.

We say that $\mathcal{A}$ accepts w if there is a sequence of configuration of $\mathcal{A}$ on w that leads $\mathcal{A}$ from the initial to a final configuration while reading w.

Remark Given a k-register data word automaton $\mathcal{A}$ and a tuple $\tau \in \mathcal{D}_\perp^k$, we can turn $\mathcal{A}$ into an automaton $\mathcal{A}(\tau)$ defined just as $\mathcal{A}$ but starting with τ as the register configuration. Such an extension does not affect the class of accepted languages, but will be useful in inductive constructions when automata need not start with all registers unassigned.

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

Lemma 1. *Let $\mathcal{A}$ be a k-register data word automaton. If $\mathcal{A}$ recognizes some word of length n, then it recognizes a word of length n that uses at most $k+1$ different data values.*

Proof. We first set some notation. We will say that two k-register assignments τ and $\overline{\tau}$ are of the same equality type if we have $\tau(i) = \tau(j)$ if and only if $\overline{\tau}(i) = \overline{\tau}(j)$, for all $i, j \leq k$. Note that this also implies that $\tau(i) \neq \tau(j)$ if and only if $\overline{\tau}(i) \neq \overline{\tau}(j)$.

We will prove a slightly more general claim, allowing our automata to start with an nonempty assignment of the registers. Let $\mathcal{A}(\tau_0) = (Q, q_0, F, T)$ be a k-register data word automaton, starting with the initial assignment τ_0 in the registers and $w = \binom{a_1}{d_1} \dots \binom{a_n}{d_n}$ a word that it accepts. This means that there is a sequence of states $q_0, q_1, \dots, q_n$, with $q_n \in F$ and a sequence of register assignments $\tau_0, \tau_1, \dots, \tau_n$ such that $(q_{i-1}, a_i, c_i) \rightarrow (I_i, q_i) \in T$, that $\tau_{i-1}, d_i \models c_i$ and τ_i is obtained from $\tau_{i-1}$by replacing all registers from I_i with d_i, for $i = 1 \dots n$.

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

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

We prove by induction on $i \leq n$ that we can define a data word w_i, of length i, such that $w_i = \binom{a_1}{d_1^i} \dots \binom{a_i}{d_i^i}$, where $a_1, \dots a_i$ are from w and $d_1^i, \dots, d_i^i$ are from S. We then show that for this w_i there is a sequence of assignments $\tau_0', \tau_1', \dots \tau_i'$ such that each τ_j' is of the same equality type as τ_j, where $j \leq i$ and it holds that $\tau_{j-1}, d_j \models c_j$, for all $j \leq i$ and each τ_j' is obtained from τ_{j-1}' by replacing all the data values from I_j by d_j. Note that this actually means that $\mathcal{A}$ goes through the same sequence of states while reading w_i as it did while reading w. But then w_n is the desired word from the statement of the lemma.

To prove this we first assume that $i = 1$. We set $\tau_0' = \tau_0$ and select $d \in S$ such that $\tau_0, d \models c_1$ (note that this is possible since we have $k+1$ values at disposal and test only for equality or inequality with a fixed set of k elements) and such that τ_1 and τ_1' are of the same equality type, where τ_1' is obtained from τ_0' by replacing all data values from I_1 by d. Again, this is possible since the original d_1 (from w) could have either been different from all data values in τ_0 or equal to some of them, a choice we can simulate with elements from S. We now set $w_1 = \binom{a_1}{d}$.

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

simply define $w_{i+1} = w_i \cdot \binom{a_{i+1}}{d}$. Note that this w_{i+1} has all the desired properties and can take $\mathcal{A}$ from q_0 to q_{i+1}.

This concludes the proof of the lemma. □

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

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

- $Q' = Q \times D_\perp^k$;
- $q_0' = (q_0, \perp^k)$;
- $F' = F \times D_\perp^k$;
- Whenever we have a transition $(q, a, c) \to (I, q')$ in T, we add the transition

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

to T if $d, \tau \models c$ and τ' is obtained from τ by putting d in positions from the set I.

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

Lemma 2. *Let D be a finite set of data values and $\mathcal{A}$ a register automaton over Σ. Then there exists a finite state automaton $\mathcal{A}_D$ over the alphabet $\Sigma \times D$ such that $w \in L(\mathcal{A}_D)$ iff $w \in L(\mathcal{A})$, for every w with data values from D. Moreover, $\mathcal{A}_D$ is of size exponential in the size of $\mathcal{A}$ and polynomial in the size of D.*

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 [13]. Although our notion of automata is slightly different than the one used there, all constructions from [13] can be easily modified to work in the setting proposed here.

Fact 1 ([13]) *1. The set of languages recognized by register automata is closed under union, intersection, concatenation and Kleene star.*
2. Languages recognized by register automata are not closed under complement.
3. Languages recognized by register automata are closed under automorphisms: that is, if $f : \mathcal{D} \to \mathcal{D}$ is an automorphism 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 $\mathcal{A}$.

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

Recall that nonemptiness problem for an automaton $\mathcal{A}$ is checking whether $L(\mathcal{A}) \neq \emptyset$.

Fact 2 ([9]) *The nonemptiness problem for register data word automata is* PSPACE*-complete.*

The lower bound will follow from Theorem 1 and Proposition 1. For the upper bound we convert our k-register automaton $\mathcal{A}$ into an NFA $\mathcal{A}_D$ over the alphabet $\Sigma \times D$ (as in the Lemma 2), where $D = \{0, \ldots, k+1\}$. We know that $\mathcal{A}_D$ recognizes all data words from $L(\mathcal{A})$ using only data values from D. By Lemma 1 and invariance under automorphisms, we know that checking $\mathcal{A}$ for nonemptiness is equivalent to checking $\mathcal{A}_D$ for nonemptiness. Using on-the-fly construction we get the desired result (note that $\mathcal{A}_D$ can not be created before checking it for nonemptiness).

The membership problem asks, for an automaton $\mathcal{A}$ and a word w, whether $w \in L(\mathcal{A})$.

Fact 3 ([21]) *The membership problem for register data word automata is* NP*-complete.*

The lower bound will follow from Theorem 1 and Proposition 2. For the upper bound it simply suffices to guess an accepting run of the automaton.

3 Regular expressions with memory

In this section we develop regular expressions capturing register automata in the same way as the usual regular expressions capture regular languages. To do this notice that register automata could be pictured as finite state automata whose transitions between states have labels of the form $a[c]\downarrow I$, where I is a set of registers. Such an automaton can move from one state to another using an arrow $a[c]\downarrow I$ if the letter it sees is a, and the data value (together with the current register assignment) satisfies the condition c. It then proceeds to the next state and updates the registers in I with the current data value. This suggests that the basic building blocks for our expressions will be expressions of the form $a[c]\downarrow I$.

Definition 2 (Expressions with memory). *Let Σ be a finite alphabet and $x_1, \ldots, x_k$ a finite set of variables. Regular expressions with memory over $\Sigma[x_1, \ldots, x_k]$ are defined inductively as follows:*

- *ε and $\emptyset$ are expressions;*
- *$a[c]\downarrow I$ is an expression; here $a \in \Sigma$, c is a condition in $\mathcal{C}_k$, and $I \subseteq \{x_1, \ldots, x_k\}$;*
- *If e, e_1, e_2 are expressions, then so are $e_1 + e_2$, $e_1 \cdot e_2$, and e^*.*

For convenience we will write just a if $I = \emptyset$ and the condition $c = \mathtt{tt}$ and similarly when only one of them can be ignored. Also, if $I = \{x\}$, we write $a[c]\downarrow x$, or $a\downarrow x$ when $c = \mathtt{tt}$, instead of $a[c]\downarrow I$.

To define the semantics, we first define what it means for an expression e over $\Sigma[x_1, \ldots x_k]$, a data word w and a tuple $\sigma \in \mathcal{D}_\perp^k$ to infer another tuple $\sigma' \in \mathcal{D}_\perp^k$, viewed as partial assignment of values to variables. We do this inductively on e.

- $(\varepsilon, w, \sigma) \vdash \sigma'$ iff $w = \varepsilon$ and $\sigma' = \sigma$.

- $(a[c]\downarrow I, w, \sigma) \vdash \sigma'$ iff $w = \binom{a}{d}$ and $\sigma, d \models c$ and σ' is obtained from σ by assigning d to each $x_i \in I$.
- $(e_1 \cdot e_2, w, \sigma) \vdash \sigma'$ iff $w = w_1 \cdot w_2$ and there exists a valuation σ'' such that $(e_1, w_1, \sigma) \vdash \sigma''$ and $(e_2, w_2, \sigma'') \vdash \sigma'$.
- $(e_1 + e_2, w, \sigma) \vdash \sigma'$ iff $(e_1, w, \sigma) \vdash \sigma'$ or $(e_2, w, \sigma) \vdash \sigma'$.
- $(e^*, w, \sigma) \vdash \sigma'$ iff
  1. $w = \varepsilon$ and $\sigma = \sigma'$, or
  2. $w = w_1 \cdot w_2$ and there exists a valuation σ'' such that $(e, w_1, \sigma) \vdash \sigma''$ and $(e^*, w_2, \sigma'') \vdash \sigma'$.

We say that a regular expression e *induces* a tuple $\sigma \in \mathcal{D}_\perp^k$ on a data word w if $(e, w, \perp^k) \vdash \sigma$. We then define $L(e)$, the language of e, as the set of all data words on which e induces some tuple σ. A regular expression with memory e is *well-formed* if every variable is bound before being used in a condition. From now on we will assume that all our expressions are well-formed.

Example 1. We now give a few examples of data word languages definable by regular expressions with memory.

1. The expression $(a\downarrow x) \cdot (b[x^{\neq}])^*$ defines the language of data words where word part reads ab^* and such that the first data value is different from all others. It binds while reading the first a, and then it proceeds checking that the letter is b and condition $x^{\neq}$ is satisfied, which is expressed by $b[x^{\neq}]$; the expression is then put in the scope of * to indicate that the number of such values is arbitrary.
2. The language of data words in which two data values are the same is given by the expression $\Sigma^* \cdot (\Sigma\downarrow x) \cdot \Sigma^* \cdot (\Sigma[x^{=}]) \cdot \Sigma^*$, where Σ is the shorthand for $a_1 + \ldots + a_l$, whenever $\Sigma = \{a_1, \ldots, a_l\}$ and $\Sigma\downarrow x$ is a shorthand for $a_1\downarrow x + \ldots + a_l\downarrow x$. It says: at some point, bind x, and then check that after one or more letters, we have the same data value.
3. The language of data words in which the last two data values occur elsewhere in the word with label a is defined by $\Sigma^* \cdot (a\downarrow x) \cdot \Sigma^* \cdot (a\downarrow y) \cdot \Sigma^* \cdot (\Sigma[x^{=}] + \Sigma[y^{=}]) \cdot (\Sigma[x^{=}] + \Sigma[y^{=}])$.

3.1 Equivalence with register automata

In this section we prove that every language recognized by register automata can also be described by a regular expression with memory and vice versa. In fact, we show a tighter connection, from which the equivalence will follow. Let $L(e, \sigma, \sigma')$ be the set of all data words w such that $(e, w, \sigma) \vdash \sigma'$, and let $L(\mathcal{A}, \sigma, \sigma')$ be the set of all data words w such that w is accepted by $\mathcal{A}(\sigma)$, and there exists an accepting run that ends with a register configuration σ'.

Theorem 1. *1. For every regular expression with memory e over $\Sigma[x_1, \ldots, x_k]$ there exists (and can be constructed in logarithmic space) a k-register data word automaton $\mathcal{A}_e$ such that $L(e, \sigma, \sigma') = L(\mathcal{A}_e, \sigma, \sigma')$ for every $\sigma, \sigma' \in \mathcal{D}_\perp^k$.*

2. For every k-register data word automaton $\mathcal{A}$ there exists (and can be constructed in exponential time) a regular expression with memory $e_\mathcal{A}$ over $x_1, \ldots, x_k$ such that $L(e_\mathcal{A}, \sigma, \sigma') = L(\mathcal{A}, \sigma, \sigma')$ for every $\sigma, \sigma' \in \mathcal{D}_\perp^k$.

The structure of the proof follows of course the standard NFA-regular expressions equivalence, cf. [24], with all the necessary adjustments to handle transitions induced by $a[c]\downarrow I$. Details can be found in the complete version of the paper. Since $L(e) = \bigcup_\sigma L(e, \perp^k, \sigma)$ and $L(\mathcal{A}) = \bigcup_\sigma L(\mathcal{A}, \perp^k, \sigma)$, we obtain:

Corollary 1. *The classes of languages of data words definable by k-register data word automata, and by regular expressions with memory over $\Sigma[x_1, \ldots, x_k]$ are the same.*

3.2 Properties of regular expressions with memory

Corollary 1 and closure properties of register automata immediately imply that languages defined by regular expressions with memory are closed under union, intersection, concatenation, Kleene star, but are *not* closed under complement.

We now turn to the nonemptiness problem, i.e., checking whether $L(\mathcal{A}) \neq \emptyset$. Since going from expressions to automata is polynomial, we get a PSPACE upper bound (see Fact 2). One can also prove a matching lower bound, by adapting techniques used in a different but related setting [16] for combined complexity bounds on query evaluation over graph databases and obtain:

Proposition 1. *The nonemptiness problem for regular expressions with memory is* PSPACE*-complete.*

Next we move to the membership problem, i.e., checking whether $w \in L(e)$. Again, since e can be translated efficiently into an equivalent automaton $\mathcal{A}_e$, Fact 3 gives an NP upper bound. We can prove a matching lower bound as well:

Proposition 2. *The membership problem for regular expressions with memory is* NP*-complete.*

Proof. For the lower bound we do a reduction from 3-SAT.

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

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

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

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

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

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

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

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

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

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

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

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

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

4 Regular expressions with equality

In this section we define yet another kind of expressions, regular expressions with equality, that will have significantly better algorithmic properties that regular expressions with memory and register automata, 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.

Originally motivation for such expressions came from graph databases, where they were used to lower combined complexity of queries that mixed data and topology. Such queries, with conditions specified by register automata, had PSPACE-complete combined complexity; with the restrictions similar to those described here, it dropped to PTIME, or to NP-complete when such queries were closed under conjunction and existential quantification [16]. These bounds are the best possible, in light of the results on regular path queries. We also argue that, although limited in expressive power, they still allow specification of interesting properties in graph or XML databases.

Definition 3 (Expressions with equality). *Let Σ be a finite alphabet. Then* regular expressions with equality *are defined by the grammar:*

$$e \;:=\; \emptyset \mid \varepsilon \mid a \mid e + e \mid e \cdot e \mid e^+ \mid e_= \mid e_{\neq} \tag{1}$$

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

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

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 $e^+ + \varepsilon$.

We now provide two examples. The expression $\Sigma^* \cdot (a \cdot \Sigma^* \cdot a)_= \cdot \Sigma^*$ denotes the language of data words that contain two a-labelled positions with the same data value. In XML this simply specifies that a is not a key. The language of data words in which the first and the last data value are different is given by $(\Sigma \cdot \Sigma^+)_{\neq}$.

4.1 Properties of regular expressions with equality

As expected regular expressions with equality will be subsumed by register automata, but unlike expressions with memory, they will be less expressive, as illustrated by the following result.

Proposition 3. *Regular expressions with equality are strictly weaker than regular expressions with memory.*

When proving this, we simply show that regular expressions with equality can be translated into register automata using an easy inductive construction. Moreover, this translation can be carried in PTIME (in fact in NLOGSPACE). To show they are strictly weaker than expressions with memory or register automata, we show that they cannot define the language of $(a{\downarrow}x) \cdot (a[x^{\neq}])^*$. To do so, we introduce another kind of automata, called weak register automata, and show that they cannot recognize that language and that they can define any language described by expressions with equality.

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 4. *Languages recognized by regular expressions with equality are not closed under intersection and complement.*

Proof sketch. Observe first that the expression $\Sigma^* \cdot (\Sigma \cdot \Sigma^+)_= \cdot \Sigma^*$ defines a 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 [13]. By Proposition 3 this implies that regular expressions with memory are not closed under complement.

To see that they are not closed under intersection we first show that the language

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

is not recognizable by any regular expression with equality. To prove this we simply try out all possible combinations of expressions that use at most three concatenated occurrences of a. Note that we can eliminate any expression with more that three as, or one that uses * (since this results in arbitrary long words), or union (since every member of the union would have to define words from this language and since we do not use constants we cannot just split the language into two or more parts). Also, no $=$ can occur in our expression (for subexpressions of length at least 2). This reduces the number of potential expressions to denote the language to finitely many possibilities, and we simply try them all.

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

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

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

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

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

To obtain fast membership and nonemptiness testing algorithms for expressions with equality, we first show how to reduce them to pushdown automata when only finite alphabets are involved.

Assume that we have a finite set D of data values. We now inductively construct PDAs $P_{e,D}$ for all regular expressions with equality e. The words recognized by these automata will be precisely the words from $L(e)$ whose data values come from D.

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

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

the first data value (here we use the new stack symbol) and compare it for equality or inequality with the last data value of the input word. The additional assumptions are here to assure that the construction works correctly. Details of the proof can be found in the full version.

Lemma 3. *The language of words accepted by each PDA $P_{e,D}$ is equal to the set of data words in $L(e)$ whose data values come from D. Moreover, the PDA $P_{e,D}$ has at most $O(|e|)$ states and $O(|e| \times (|D|^2 + |e|))$ transitions, and can be constructed in polynomial time.*

From this and Lemma 1 it is easy to obtain the following.

Theorem 2. *The nonemptiness problem for regular expressions with equality is in* PTIME.

To see this, take an arbitrary expression with equality e and convert it to a n-register data word automaton $\mathcal{A}$ that recognizes the same language. From the translation, we know that n will be at most the number of times $=$ and $\neq$ appear in e. Now do the construction from Lemma 3 for e and $D = \{0, 1, \ldots, n+1\}$ to obtain a PDA $P_{e,D}$. Proposition 3 and Lemma 1 now imply that checking if $L(e) \neq \emptyset$ is equivalent to checking $P_{e,D}$ for nonemptiness. Since this automaton is of polynomial size, we can check it for nonemptiness in PTIME thus obtaining the desired result.

Proposition 5. *The membership problem for regular expressions with equality is in* PTIME.

As in the proof of Theorem 2, we construct a PDA $P_{e,D}$ for e and $D = \{0, 1, \ldots, n\}$, where n is the length of the input word w. By invariance under automorphisms 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. The correctness of this algorithm follows from Lemma 3.

It is natural to ask whether NFAs could not have been used instead of pushdown automata. The answer is that they can be used to capture languages of data words described by regular expressions with equality over a finite set of data values, but the cost is necessarily exponential, and hence we cannot possible use them to derive Theorem 2. That is, we can first show:

Proposition 6. *For every regular expression with equality e over the alphabet Σ and a finite set D of data values there exists an NFA $\mathcal{A}_{e,D}$, of the size exponential in $|e|$, recognizing precisely those data words from $L(e)$ that use data values from D.*

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

state we test that the current data value is different (or the same) to the one corresponding to this copy of the original automaton. This is done in a manner analogous to the one used in the proof of Proposition 3. □

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

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

1. $x_i \cdot y_i \in L$, for every $i = 1, \ldots n$, and
2. $x_i \cdot y_j \notin L$, for $1 \leq i, j \leq n$ and $i \neq j$.

Then any NFA accepting L has at least n states.

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

Next we define the expressions e_n inductively as follows:

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

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

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

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

Since these pairs satisfy the above assumptions 1) and 2), we conclude, using the result of [11], that any NFA recognizing $L(e_n)$ has at least $O(2^{|e_n|})$ states, so no regular expression describing it can be of length polynomial in $|e_n|$.

5 Conclusions and future work

Here we addressed the problem of finding analogs of regular expressions for register automata, and explored their language-theoretic properties. We also defined an expressive subclass with good algorithmic properties. In the future we would like to try and find an intermediate class of expressions that could be used to recognize a larger class of languages than regular expressions with equality, but still retain low complexity of nonemptiness and membership checking. We would also like to explore how these new classes of expressions behave as query languages in graph database models. Since language nonemptiness is closely related to query evaluation in that context we are hopeful to obtain fast and expressive query languages based on these new classes of expressions.

Acknowledgment We would like to thank Juan Reutter and Tony Tan for helpful comments during the preparation of this paper. Work partially supported by EPSRC grant G049165 and FET-Open Project FoX, grant agreement 233599.

References

1. R. Angles, C. Gutiérrez. 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 *PODS'10*, pages 3–14.
3. M. Benedikt, C. Ley, G. Puppis. Automata vs. logics on data words. In *CSL 2010*, pages 110–124.
4. M. Bojanczyk, P. Parys. XPath evaluation in linear time. In *PODS'08*, pages 241-250.
5. M. Bojanczyk, C. David, A. Muscholl, T. Schwentick, L. Segoufin. Two-variable logic on words with data. *ACM TOCL* 12(4): (2011).
6. M. Bojanczyk, S. Lasota. An extension of data automata that captures XPath. In *LICS 2010*, pages 243–252.
7. D. Calvanese, G. de Giacomo, M. Lenzerini, M. Y. Vardi. Rewriting of regular expressions and regular path queries. *JCSS*, 64(3):443–465, 2002.
8. T. Colcombet, C. Ley, G. Puppis. On the use of guards for logics with data. *MFCS 2011*, pages 243–255.
9. S. Demri, R. Lazic. LTL with the freeze quantifier and register automata. *ACM TOCL* 10(3): (2009).
10. D. Figueira. Satisfiability of downward XPath with data equality tests. *PODS'09*, 197-206.
11. I. Glaister, J. Shallit. A lower bound technique for the size of nondeterministic finite automata. *IPL* 59:75-77, 1996.
12. O. Grumberg, O. Kupferman, S. Sheinvald. Variable automata over infinite alphabets. In *LATA'10*, pages 561–572.
13. M. Kaminski and N. Francez. Finite memory automata. *Theoretical Computer Science*, 134(2):329-363, 1994.
14. M. Kaminski and T. Tan. Regular expressions for languages over infinite alphabets. *Fundam. Inform.*, 69(3):301-318, 2006.
15. L. Libkin. Logics for unranked trees: an overview. *Logical Methods in Computer Science* 2(3): (2006).
16. L. Libkin, D. Vrgoč. Regular path queries on graphs with data. In *ICDT 2012*, to appear.
17. M. Marx. Conditional XPath. *ACM TODS*, 30 (2005), 929–959.
18. A. O. Mendelzon, P. T. Wood. Finding regular simple paths in graph databases. *SIAM J. Comput*, 24(6):1235-1258 (1995).
19. F. Neven. Automata theory for XML researchers. *SIGMOD Record*, 31(3):39?46, 2002.
20. F. Neven, Th. Schwentick, V. Vianu. Finite state machines for strings over infinite alphabets. *ACM TOCL* 5(3):403-435 (2004).
21. H. Sakamoto and D. Ikeda., Intractability of decision problems for finite-memory automata. *Theor. Comput. Sci*. 231, 2, 297-308, 2000.
22. T. Schwentick. Automata for XML – A survey. *JCSS* 73(3): 289-315 (2007).
23. L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In *CSL'06*, pages 41-57.
24. M. Sipser, Introduction to the Theory of Computation. PWS Publishing, 1997.
25. T. Tan. Graph reachability and pebble automata over infinite alphabets. In *LICS 2009*, pages 157–166.