Regular Expressions with Binding over Data Words for Querying Graph Databases

Leonid Libkin[1], Tony Tan[2], and Domagoj Vrgoč[1]

[1] University of Edinburgh
libkin@inf.ed.ac.uk, domagoj.vrgoc@ed.ac.uk
[2] Hasselt University and Transnational University of Limburg
tony.tan@uhasselt.be

Abstract. Data words assign to each position a letter from a finite alphabet and a data value from an infinite set. Introduced as an abstraction of paths in XML documents, they recently found applications in querying graph databases as well. Those are actively studied due to applications in such diverse areas as social networks, semantic web, and biological databases. Querying formalisms for graph databases are based on specifying paths conforming to some regular conditions, which led to a study of regular expressions for data words.

Previously studied regular expressions for data words were either rather limited, or had the full expressiveness of register automata, at the expense of a quite unnatural and unintuitive binding mechanism for data values. Our goal is to introduce a natural extension of regular expressions with proper bindings for data values, similar to the notion of freeze quantifiers used in connection with temporal logics over data words, and to study both language-theoretic properties of the resulting class of languages of data words, and their applications in querying graph databases.

1 Introduction

Data words, unlike the usual words over finite alphabet, assign to each position both a letter from a finite alphabet and an element of an infinite set, referred to as a data value. An example of a data word is $\binom{a}{1}\binom{b}{2}\binom{a}{3}\binom{b}{1}$. This is a data word over the finite alphabet $\{a, b\}$, with data elements coming from an infinite domain, in this case, $\mathbb{N}$. Investigations of data words picked up recently due to their importance in the study of XML documents. Those are naturally modeled as ordered unranked trees in which every node has both a label and a datum (these are referred to as data trees). Data words then model paths in data trees, and as such are essential for investigations of many path-based formalisms for XML, for instance, its navigational query language XPath. We refer the reader to [7, 14, 30, 31] for recent surveys.

While the XML data format dominated the data management landscape for a while, primarily in the 2000s, over the past few years the focus started shifting towards the *graph* data model. Graph-structured data appears naturally in a

M.-P. Béal and O. Carton (Eds.): DLT 2013, LNCS 7907, pp. 325–337, 2013.

variety of applications, most notably social networks and the Semantic Web (as it underlies the RDF format). Its other applications include biology, network traffic, crime detection, and modeling object-oriented data [13, 21, 24, 26–29]. Such databases are represented as graphs in which nodes are objects and the edge labels specify relationships between them; see [1, 4] for surveys.

Just as in the case of XML, a crucial building block in queries against graph data deals with properties of paths in them. The most basic formalism is that of *regular path queries*, or *RPQs*, which select nodes connected by a path described by a regular language over the labeling alphabet [11]. There are multiple extensions with more complex patterns, backward navigation, regular relations over paths, and non-regular features [3, 5, 6, 8–10]. In real applications we deal with both navigational information and data, so it is essential that we look at properties of paths that also describe how data values change along them. Since such paths (as we shall explain later) are just data words, it becomes necessary to provide expressive and well-behaved mechanisms for describing languages of data words.

One of the most commonly used formalisms for describing the notion of regularity for data words is that of *register automata* [19]. These extend the standard NFAs with registers that can store data values; transitions can compare the currently read data value with values stored in registers.

However, register automata are not convenient for specifying properties – ideally, we want to use regular expressions to define languages. These have been looked at in the context of data words (or words over infinite alphabets), and are based on the idea of using *variables* for binding data values. An initial attempt to define such expressions was made in [20], but it was very limited. Another formalism, called *regular expressions with memory*, was shown to be equivalent to register automata [22, 23]. At the first glance, they appear to be a good formalism: these are expressions like $a \downarrow_x (a[x^=])^*$ saying: read letter a, bind data value to x, and read the rest of the data word checking that all letters are a and the data values are the same as x. This will define data words $\binom{a}{d} \cdots \binom{a}{d}$ for some data value d. This is reminiscent of freeze quantifiers used in connection with the study of data word languages [12].

The serious problem with these expressions, however, is the *binding* of variables. The expression above is fine, but now consider the following expression: $a \downarrow_x (a[x^=]a \downarrow_x)^* a[x^=]$. This 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, this feature was essential for capturing register automata [22]. So natural questions arise:

– Can we define regular expressions for data words that use the acceptable scope/binding policies for variables? Such expressions will be more declarative than procedural, and more appropriate for being used in queries.
– Do these fall short of the full power of register automata?
– What are their basic properties, and what is the complexity of querying graph data with such expressions?

Contributions. Our main contribution is to define a new formalism of *regular expressions with binding*, or REWBs, to study its properties, and to show how it can be used in the context of graph querying. The binding mechanism of REWBs follows the standard scoping rules, and is essentially the same as in LTL extensions with freeze quantifiers [12]. We also look at some subclasses of REWBs based on the types of conditions one can use: in *simple* REWBs, each condition involves at most one variable (all those shown above were such), and in *positive* REWBs, negation and inequality cannot be used in conditions.

We show that the class of languages defined by REWBs is strictly contained in the class of languages defined by register automata. The separating example is rather intricate, and indeed it appears that for most reasonable languages one can think of, if they are definable by register automata, they would be definable by REWBs as well. At the same time, REWBs lower the complexity of some key computational tasks related to languages of data words. For instance, nonemptiness is PSPACE-complete for register automata [12], but we show that it is NP-complete for REWBs (and trivializes for simple and positive REWBs).

We consider the containment and universality problems for REWBs. In general they are undecidable, even for simple REWBs. However, the problem becomes decidable for positive REWBs.

We look at applications of REWBs in querying graph databases. The problem of query evaluation is essentially checking whether the intersection of two languages of data words is nonempty. We use this to show that the complexity of query evaluation is PSPACE-complete (note that it is higher than the complexity of nonemptiness alone); for a fixed REWB, the complexity is tractable.

At the end we also sketch some results concerning a model of data word automaton that uses variables introduced in [16]. We also comment on how these can be combined with register automata to obtain a language subsuming all the previously used ones while still retaining good query evaluation bounds.

Organization. We define data words and data graphs in Section 2. In Section 3 we introduce our notion of regular expression with binding (REWB) and study their nonemptiness and universality problems in Section 4 and Section 5, respectively. In Section 6 we study REWBs as a graph database query language and in Section 7 we consider some possible extensions that could be useful in graph querying. Due to space limitations, complete proofs of all the results are in the appendix.

2 Data Words and Data Graphs

Let Σ be a finite alphabet and $\mathcal{D}$ a countable infinite set of data values. A **data word** is simply a finite string over the alphabet $\Sigma \times \mathcal{D}$. That is, in each position

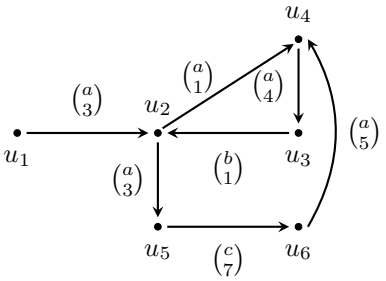
a data word carries a letter from Σ and a data value from $\mathcal{D}$. We will denote data words by $\binom{a_1}{d_1}\dots\binom{a_n}{d_n}$, where $a_i \in \Sigma$ and $d_i \in \mathcal{D}$.

A **data graph** (over Σ) is pair $G = (V, E)$, where

- V is a finite set of nodes;
- $E \subseteq V \times \Sigma \times \mathcal{D} \times V$ is a set of edges where each edge contains a label from Σ and a data value from $\mathcal{D}$.

We write $V(G)$ and $E(G)$ to denote the set of nodes and edges of G, respectively. An edge e from a node u to a node u' is written in the form $(u, \binom{a}{d}, u')$, where $a \in \Sigma$ and $d \in \mathcal{D}$. We call a the label of the edge e and d the data value of the edge e. We write $\mathcal{D}(G)$ to denote the set of data values in G.

The following is an example of a data graph, with nodes $u_1, \dots, u_6$ and edges $(u_1, \binom{a}{3}, u_2)$, $(u_3, \binom{b}{1}, u_2)$, $(u_2, \binom{a}{3}, u_5)$, $(u_6, \binom{a}{5}, u_4)$, $(u_2, \binom{a}{1}, u_4)$, $(u_4, \binom{a}{4}, u_3)$ and $(u_5, \binom{c}{7}, u_6)$.



A path from a node v to a node v' in G is a sequence

$$\pi = v_1 \begin{pmatrix} a_1 \\ d_1 \end{pmatrix} v_2 \begin{pmatrix} a_2 \\ d_2 \end{pmatrix} v_3 \begin{pmatrix} a_3 \\ d_3 \end{pmatrix} \cdots v_n \begin{pmatrix} a_n \\ d_n \end{pmatrix} v_{n+1}$$

such that each $(v_i, \binom{a_i}{d_i}, v_{i+1})$ is an edge for each $i \leq n$, and $v_1 = v$ and $v_{n+1} = v'$.

A path π defines a data word $w(\pi) = \binom{a_1}{d_1}\binom{a_2}{d_2}\binom{a_3}{d_3}\cdots\binom{a_n}{d_n}$.

Remark. Note that we have chosen a model in which labels and data values appear in edges. Of course other variations are possible, for instance labels appearing in edges and data values in nodes. All of these easily simulate each other, very much in the same way as one can use either labeled transitions systems or Kripke structures as models of temporal or modal logic formulae. In fact both models – with labels in edges and labels in nodes – have been considered in the context of semistructured data and, at least from the point of view of their expressiveness, they are viewed as equivalent. Our choice is dictated by the ease of notation primarily, as it identifies paths with data words.

3 Regular Expressions with Binding

We now define regular expressions with binding for data words. As explained already, expressions with variables for data words were previously defined in [23]

but those were really designed to mimic the transitions of register automata, and had very procedural, rather than declarative flavor. Here we define them using proper scoping rules.

Variables will store data values; those will be compared with other variables using conditions. To define them, 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:

$$c \;:=\; \top \mid \bot \mid x_i^{=} \mid x_i^{\neq} \mid c \wedge c \mid c \vee c \mid \neg c, \qquad 1 \leq i \leq k.$$

The satisfaction of a condition is defined with respect to a data value $d \in \mathcal{D}$ and a (partial) valuation $\nu : \{x_1, \ldots, x_k\} \to \mathcal{D}$ of variables as follows:

- $d, \nu \models \top$ and $d, \nu \not\models \bot$;
- $d, \nu \models x_i^{=}$ iff $d = \nu(x_i)$;
- $d, \nu \models x_i^{\neq}$ iff $d \neq \nu(x_i)$;
- the semantics for Boolean connectives $\vee, \wedge$, and $\neg$ is standard.

Next we define regular expressions with binding.

Definition 1. *Let Σ be a finite alphabet and $\{x_1, \ldots, x_k\}$ a finite set of variables.* Regular expressions with binding *(REWB) over $\Sigma[x_1, \ldots, x_k]$ are defined inductively as follows:*

$$r \;:=\; \varepsilon \;\mid\; a \;\mid\; a[c] \;\mid\; r + r \;\mid\; r \cdot r \;\mid\; r^* \;\mid\; a \downarrow_{x_i} (r) \qquad (1)$$

where $a \in \Sigma$ and c 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 $a[c]$ all variables occurring in c 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.

A valuation on the variables $x_1, \ldots, x_k$ is a partial function $\nu : \{x_1, \ldots, x_k\} \mapsto \mathcal{D}$. 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 $x \neq x_i$. Likewise, we write $\nu[\bar{x} \leftarrow \bar{d}]$ for a simultaneous substitution of values from $\bar{d} = (d_1, \ldots, d_l)$ for variables $\bar{x} = (x_1, \ldots, x_l)$. Also notation $\nu(\bar{x}) = \bar{d}$ means that $\nu(x_i) = d_i$ for all $i \leq l$.

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 $r = a$ and $a \in \Sigma$, then $L(r, \nu) = \{\binom{a}{d} \mid d \in \mathbb{N}\}$.
- If $r = a[c]$, then $L(r, \nu) = \{\binom{a}{d} \mid d, \nu \models c\}$.
- If $r = r_1 + r_2$, then $L(r, \nu) = L(r_1, \nu) \cup L(r_2, \nu)$.

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

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

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

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

Register Automata and Expressions with Memory. As mentioned earlier, *register automata* extend NFAs with the ability to store and compare data values. Formally, an automaton with k registers is $\mathcal{A} = (Q, q_0, F, T)$, 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, $I \subseteq \{1, \ldots, k\}$, 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 $\tau : \{1, \ldots, k\} \to \mathcal{D}$, 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. For more details on register automata we refer reader to [19, 23].

Expressions introduced in [22] had a similar syntax but rather different semantics. They were built using $a \downarrow_x$, concatenation, union and Kleene star. That is, no binding was introduced with $a \downarrow_x$; rather it directly matched the operation of putting a value in a register. In contrast, we 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. This corresponds to the standard binding policies in logic, or in programs.

Example 1. We list several examples of languages expressible with our expressions. In all cases below we have a singleton alphabet $\Sigma = \{a\}$.

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

Note that in REWBs in the above example the conditions are very simple: they are either $x^{=}$ or $x^{\neq}$. We will call such expressions *simple* REWBs.

We shall also consider *positive* REWBs where negation and inequality are disallowed in conditions. That is, all the conditions c are constructed using the following syntax: $c := \top \mid x_i^{=} \mid c \wedge c \mid c \vee c$,, where $1 \leq i \leq k$.

We finish this section by showing that REWBs are strictly weaker than register automata (i.e., proper binding of variables has a cost – albeit small – in terms of expressiveness).

Theorem 1. *The class of languages defined by REWBs is strictly contained in the class of languages accepted by register automata.*

That the class of languages defined by REWBs is contained in the class of languages defined by register automata can be proved by using a similar inductive construction as in [22, Proposition 5.3]. The idea behind the construction of the separating example follows the intuition that defining scope of variables restricts the power of the language, compared to register automata where once stored, the value remains in the register until rewritten. As the proof is rather technical and lengthy, we present it in the appendix.

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

4 The Nonemptiness Problem

We now look at the standard language-theoretic problem of nonemptiness:

	Nonemptiness for REWBs
Input:	A REWB r over $\Sigma[x_1, \ldots, x_k]$.
Task:	Decide whether $L(r) \neq \emptyset$.

More generally, one can ask if $L(r, \nu) \neq \emptyset$ for a REWB r and a compatible valuation ν.

Recall that for register automata, the nonemptiness problem is Pspace-complete [12] (and the same bound applied to regular expressions with memory [23]). Introducing proper binding, we lose little expressiveness and yet can lower the complexity.

Theorem 2. *The nonemptiness problem for REWBs is* NP*-complete.*

The proof is in the appendix. Note that for simple and positive REWBs the problem trivializes.

Proposition 1.
- *For every simple REWB r over $\Sigma[x_1, \ldots, x_k]$, and for every valuation ν compatible with r, we have $L(r, \nu) \neq \emptyset$.*
- *For every positive REWB r over $\Sigma[x_1, \ldots, x_k]$, there is a valuation ν such that $L(r, \nu) \neq \emptyset$.*

5 Containment and Universality

We now turn our attention to language containment. That is we are dealing with the following problem:

Containment for REWBs	
Input:	Two REWBs r_1, r_2 over $\Sigma[x_1, \ldots, x_k]$.
Task:	Decide whether $L(r_1) \subseteq L(r_2)$.

When r_2 is a fixed expression denoting all data words, this is the universality problem. We show that both are undecidable.

In fact, we show a stronger statement, that *universality* of simple REWBs that use just a single variable is already undecidable.

Universality for one-variable REWBs	
Input:	An REWB r over $\Sigma[x]$.
Task:	Decide whether $L(r) = (\Sigma \times \mathcal{D})^*$.

Theorem 3. Universality for one-variable REWBs *is undecidable. In particular, containment for REWBs is undecidable too.*

While restriction to simple REWBs does not make the problem decidable, the restriction to positive REWBs does: as is often the case, static analysis tasks become easier without negation.

Theorem 4. *The containment problem for positive REWBs is decidable.*

Proof. It is rather straightforward to show that any positive REWB can be converted into a register automaton without inequality [20]. The decidability of the language containment follows from the fact that the containment problem for register automata without inequality is decidable [32].

6 REWBs as a Query Language for Data Graphs

Standard mechanisms for querying graph databases are based on *regular path queries*, or RPQs: those select nodes connected by a path belonging to a given regular language [4, 9–11]. For data graphs, we follow the same idea, but now paths are specified by REWBs, since they contain data. In this section we study the complexity of this querying formalism.

We first explain how the problem of query evaluation can be cast as a problem of checking nonemptiness of language intersection.

Note that a data graph G can be viewed as an automaton, generating data words. That is, given a data graph $G = (V, E)$, and a pair of nodes s, t, we let $\mathcal{L}(G, s, t)$ be $\{w(\pi) \mid \pi \text{ is a path from } s \text{ to } t \text{ in } G\}$; this is a set of data words.

Let $r(\bar{x})$ be a REWB over $\Sigma[x_1, \ldots, x_k]$. For ν compatible with r, we let $\mathcal{L}(G, s, t, r, \nu)$ be $\mathcal{L}(G, s, t) \cap \mathcal{L}(r, \nu)$. Then for a graph $G = (V, E)$, we define the

answer to r over G as the set $\mathcal{Q}(r, G)$ of triples $(s, t, \bar{d}) \in V \times V \times \mathcal{D}^k$, such that $\mathcal{L}(G, s, t, r, \nu[\bar{x} \leftarrow \bar{d}])) \neq \emptyset$. In other words, there is a path π in G from s to t such that $w(\pi) \in L(r, \nu)$, where $\nu(\bar{x}) = \bar{d}$.

If r is a closed REWB, we do not need a valuation in the above definition. That is, $\mathcal{Q}(r, G)$ is the set of pairs of nodes (s, t) such that $\mathcal{L}(G, s, t) \cap \mathcal{L}(r) \neq \emptyset$, i.e., there is a path π in G from s to t such that $w(\pi) \in L(r)$.

In what follows we are interested in the query evaluation and query containment problems. For simplicity we will work with closed REWBs only. We start with query evaluation.

	Query Evaluation for REWB
Input:	A data graph G, two nodes $s, t \in V(G)$ and a REWB r.
Task:	Decide whether $(s, t) \in \mathcal{Q}(r, G)$.

Note that in this problem, both the data graph and the query, given by r, are inputs; this is referred to as the *combined complexity* of query evaluation. If the expression r is fixed, we are talking about *data complexity*.

Recall that for the usual graphs (without data), the combined complexity of evaluating RPQs is polynomial, but if conjunctions of RPQs are taken, it goes up to NP (and could be NP-complete, in fact [10, 11]). When we look at data graphs and specify paths with register automata, combined complexity jumps to Pspace-complete [22].

However, we have seen that REWBs are less expressive than register automata, so perhaps a lower NP bound would apply to them? One way to try to do it is to find a polynomial bound on the length of a minimal path witnessing a REWB in a data graph. The next proposition shows that this is impossible, since in some cases the shortest witnessing path will be exponentially long, even if the REWB uses only one variable.

Proposition 2. *Let $\Sigma = \{\$, ¢, a, b\}$ be a finite alphabet. There exists a family of data graphs $\{G_n(s, t)\}_{n>1}$ with two distinguished nodes s and t, and a family of closed REWBs $\{r_n\}_{n>1}$ such that*

- *each $G_n(s, t)$ is of size $O(n)$;*
- *each r_n is a closed REWB over $\Sigma[x]$ of length $O(n)$; and*
- *every data word in $\mathcal{L}(G_n, s, t, r_n)$ is of length $\Omega(2^{\lfloor n/2 \rfloor})$.*

The proof of this is rather involved and can be found in the appendix.

Next we describe the complexity of the query evaluation problem. It turns out that it matches that for register automata.

Theorem 5.
- *The complexity of query evaluation for REWB is* Pspace*-complete.*
- *For each fixed r, the complexity of query evaluation for REWB is in* NLogspace.

In other words, the combined complexity of queries based on REWBs is Pspace-complete, and their data complexity is in NLogspace (and of course it can

be NLogspace-complete even for very simple expressions, e.g., Σ^*, which just expresses reachability). Note that the combined complexity is acceptable (it matches, for example, the combined complexity of standard relational query languages such as relational calculus and algebra), and that data complexity is the best possible for a language that can express the reachability problem.

We prove Pspace membership by showing how to transform REWBs into regular expressions when only finitely many data values are considered. Since the expression in question is of length exponential in the size of the input, standard on-the-fly construction of product with the input graph (viewed as an NFA) gives us the desired bound. Details of this construction, as well as the proof of hardness, can be found in the appendix. The same proof, for a fixed r, gives us the bound for data complexity.

Note that the upper bound follows from the connection with register automata. In order to make our presentation self contained we opted to present a different proof in the appendix.

By examining the proofs of Theorem 5 and Theorem 3 we observe that lower bounds already hold for both simple and positive REWBs. That is we get the following.

Corollary 1. *The following holds for simple REWBs.*

- *Combined complexity of simple (or positive) REWB queries is* Pspace*-complete.*
- *Data complexity of simple (or positive) REWB queries is* NLogspace*-complete.*

Another important problem in querying graphs is query containment. In general, the query containment problem asks, for two REWBs r_1, r_2 over $\Sigma[x_1, \ldots, x_k]$, whether $\mathcal{Q}(r_1, G) \subseteq \mathcal{Q}(r_2, G)$ for every data graph G. For REWB-based queries we look at, this problem is easily seen to be equivalent to language containment. Using this fact and the results of Section 5 we obtain the following.

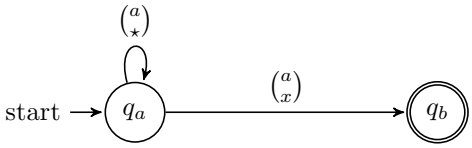
Corollary 2. *Query containment is undecidable for REWBs and simple REWBs. It becomes decidable if we restrict our queries to positive REWBs.*

7 Conclusions and Other Models

After conducting an extensive study of their language-theoretic properties and their ability to query graph data we conclude that REWBs can serve as a highly expressive language that still retains good query evaluation properties. Although weaker than register automata and their expression counterpart – regular expressions with memory, REWBs come with a more natural and declarative syntax and have a lower complexity of some language-theoretic properties such as nonemptiness. They also complete a picture of expressions that relate to register automata – a question that often came up in the discussions about the connection of regular expressions with memory (REMs) and register automata [22, 23], as they can be seen as a natural restriction of REMs with proper scoping rules.

As we have seen, both in this paper and in previous work on graph querying, all of the considered formalisms have a combined complexity of query evaluation that is either a low degree polynomial, or PSPACE-complete. A natural question to ask is if there is a formalism whose combined complexity lies between these two classes.

An answer to this can be given using a model of automata that extends NFAs in a similar way that REWBs extend regular expressions – by allowing usage of variables. These automata, called *variable automata*, were introduced in [16] and although originally defined for words over an infinite alphabet, they can easily be modified to handle data words. Intuitively, they can be viewed as NFAs with a guess of data values to be assigned to variables, with the run of the automaton verifying correctness of the guess. An example of a variable automaton recognizing the language of all words where the last data value is different from all others is given in the following image.



Here we observe that variable automata use two sorts of variables – an ordinary bound variable x that is assigned a unique value, and a special free variable $\star$, whose every occurrence is assigned a value different from the ones assigned to the bound variables.

It can be show that variable automata, used as a graph querying formalism, have NP-complete combined complexity of query evaluation and that their deterministic subclass [16] has coNP query containment. Due to space limitations we defer the technical details of these results to the appendix.

The somewhat synthetic nature of variable automata and their usage of the free variable makes them incomparable with REWBs and register automata, as the example above demonstrates. A natural question then is whether there is a model that encompasses both and still retains the same good query evaluation bounds. It can be shown that by allowing variable automata to use the full power of registers we get a model that subsumes all of the previously studied models and whose combined complexity is no worse that the one of register automata. This approach, albeit in a limited form, was already proposed in e.g. [15]. The details of the construction can be found in the appendix.

Acknowledgement. The second author acknowledges the generous financial support of FWO, under the scheme FWO Pegasus Marie Curie fellowship. The first and the third author were partially supported by the EPSRC grants J015377 and G049165.

References

1. Abiteboul, S., Buneman, P., Suciu, D.: Data on the Web: From Relations to Semistructured Data and XML. Morgan Kauffman (1999)
2. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
3. Abiteboul, S., Vianu, V.: Regular path queries with constraints. JCSS 58, 428–452 (1999)
4. Angles, R., Gutiérrez, C.: Survey of graph database models. ACM Comput. Surv. 40(1) (2008)
5. Barceló, P., Figueira, D., Libkin, L.: Graph logics with rational relations and the generalized intersection problem. In: LICS (2012)
6. Barceló, P., Libkin, L., Lin, A.W., Wood, P.: Expressive languages for path queries over graph-structured data. ACM TODS 37(4) (2012)
7. Bojanczyk, M.: Automata for Data Words and Data Trees. In: RTA, pp. 1–4 (2010)
8. Calvanese, D., de Giacomo, G., Lenzerini, M., Vardi, M.Y.: Containment of conjunctive regular path queries with inverse. In: KR 2000, pp. 176–185 (2000)
9. Calvanese, D., de Giacomo, G., Lenzerini, M., Vardi, M.Y.: Rewriting of regular expressions and regular path queries. JCSS 64(3), 443–465 (2002)
10. Consens, M.P., Mendelzon, A.O.: GraphLog: a visual formalism for real life recursion. In: PODS 1990, pp. 404–416 (1990)
11. Cruz, I., Mendelzon, A., Wood, P.: A graphical query language supporting recursion. In: SIGMOD 1987, pp. 323–330 (1987)
12. Demri, S., Lazić, R.: LTL with the freeze quantifier and register automata. ACM TOCL 10(3) (2009)
13. Fan, W.: Graph pattern matching revised for social network analysis. In: ICDT 2012, pp. 8–21 (2012)
14. Figueira, D.: Reasoning on words and trees with data. PhD thesis (2010)
15. Figueira, D.: Alternating register automata on finite words and trees. Logical Methods in Computer Science 8(1) (2012)
16. Grumberg, O., Kupferman, O., Sheinvald, S.: Variable automata over infinite alphabets. In: Dediu, A.-H., Fernau, H., Martín-Vide, C. (eds.) LATA 2010. LNCS, vol. 6031, pp. 561–572. Springer, Heidelberg (2010)
17. Grumberg, O., Kupferman, O., Sheinvald, S.: Variable automata over infinite alphabets (2011) (manuscript)
18. Gutierrez, C., Hurtado, C., Mendelzon, A.: Foundations of semantic Web databases. J. Comput. Syst. Sci. 77(3), 520–541 (2011)
19. Kaminski, M., Francez, N.: Finite-memory automata. TCS 134(2), 329–363 (1994)
20. Kaminski, M., Tan, T.: Regular expressions for languages over infinite alphabets. Fundamenta Informaticae 69(3), 301–318 (2006)
21. Leser, U.: A query language for biological networks. Bioinformatics 21(suppl. 2), 33–39 (2005)
22. Libkin, L., Vrgoč, D.: Regular path queries on graphs with data. In: ICDT 2012, pp. 74–85 (2012)
23. Libkin, L., Vrgoč, D.: Regular expressions for data words. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18 2012. LNCS, vol. 7180, pp. 274–288. Springer, Heidelberg (2012)
24. Milo, R., Shen-Orr, S., et al.: Network motifs: simple building blocks of complex networks. Science 298(5594), 824–827 (2002)

25. Neven, F., Schwentick, T., Vianu, V.: Finite state machines for strings over infinite alphabets. ACM TOCL 5(3), 403–435 (2004)
26. Olken, F.: Graph data management for molecular biology. OMICS 7, 75–78 (2003)
27. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM TODS 34(3), 1–45 (2009)
28. Ronen, R., Shmueli, O.: SoQL: a language for querying and creating data in social networks. In: ICDE 2009, pp. 1595–1602 (2009)
29. San Martín, M., Gutierrez, C.: Representing, querying and transforming social networks with RDF/SPARQL. In: Aroyo, L., et al. (eds.) ESWC 2009. LNCS, vol. 5554, pp. 293–307. Springer, Heidelberg (2009)
30. Schwentick, T.: A Little Bit Infinite? On Adding Data to Finitely Labelled Structures. In: STACS 2008, pp. 17–18 (2008)
31. Segoufin, L.: Automata and logics for words and trees over an infinite alphabet. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 41–57. Springer, Heidelberg (2006)
32. A. Tal. Decidability of Inclusion for Unification Based Automata. M.Sc. thesis (in Hebrew), Technion (1999)