Filip Mazowiecki

Uniwersytet Warszawski

Regular tree pattern queries and datalog

Praca semestralna nr 3

(semestr letni 2012/13)

Opiekun pracy: Filip Murlak

Regular tree pattern queries and datalog

Filip Mazowiecki
University of Warsaw

Filip Murlak
University of Warsaw

Adam Witkowski
University of Warsaw

ABSTRACT

Among various recursive query languages, datalog is of particular interest. Unfortunately, most classical static analysis problems for datalog are undecidable over words and trees, with the exception of satisfiability and limited variants of containment. Recently, even these problems were shown to be undecidable over data trees.

In practical applications however, one can often work under additional restrictions, which can potentially give decidability or even tractability of the considered problems. Rather than looking at generic restrictions, we focus on a concrete setting introduced for the purpose of testing equivalence of ActiveXML systems. This setting involves two main restrictions: data trees are unlabelled but queries can use data constants (equivalently, the labelling alphabet is infinite but queries can test labels for equality); queries have a restricted form of regular tree patterns (RTPQs), which are essentially regular expressions over the set of tree patterns using child and descendant axes and data equalities. We investigate how these restrictions affect the decidability of the classical static analysis problems.

We show that RTPQs extended with parent and ancestor axes are equivalent to linear monadic datalog. Using this result we prove that containment of linear monadic datalog programs in unions of conjunctive queries is undecidable even over unlabelled data trees, thus strengthening known undecidability bound. In contrast, the second restriction actually gives decidability: we show that containment of RTPQs without parent and ancestor axes is EXPSPACE-complete. This solves an open problem in equivalence testing of ActiveXML systems.

1. INTRODUCTION

Classical query languages, like SQL, are essentially fragments of first order logic (FO) extended with basic aggregation. For some kinds of data these languages are sufficient to express typical queries: finding the employee with maximal salary, or checking if each order was served within given time limit does not pose problems. But some applications typically involve queries inexpressible in first order logic, e.g., find all people supervised (not necessarily directly) by a given manager in a hierarchically structured company, or check if there is a train connection from London to Naples, or find all people that received money originating in the company "Dirty Money Laundry". The feature missing in FO, needed to express such queries, is *recursion*.

First order logic can be easily extended with various forms of recursion: from transitive closure operators to much more powerful fixpoint operators. One of the most prominent logics with recursion is datalog, obtained by adding fixpoint operator to unions of conjunctive queries (positive existential first order formulae). Datalog originated as a declarative programming language, but later found many applications in databases as a query language. Unfortunately, with increased expressive power comes high complexity or even undecidability of basic properties of queries. Classical static analysis problems of containment and equivalence are undecidable [15]. It is also undecidable if a given datalog program is equivalent to some non-recursive datalog program [11], and equivalence to a given non-recursive program is 3-EXPTIME-complete [9]. Recently, it was shown that when the class of considered structures is restricted to words or trees, even satisfiability is undecidable [2].

Since these problems are subtasks in important data management tasks like query minimization, data integration, or data exchange, the negative results for full datalog fuel interest in restrictions [4, 6, 7]. Two important restrictions include *monadic* programs, where only unary extensional predicates are allowed, and *linear* programs, where only one intensional predicate can be used on the right-hand side of each rule.

Over trees, containment is decidable for monadic datalog programs that do not use descendant [12]. But this decidability result does not carry over to the setting of data trees, where each node carries a label form a finite alphabet and a data value from an infinite data domain, and queries are allowed to test data values for equality. Over data trees, already containment of linear monadic datalog programs in unions of conjunctive queries is undecidable, and positive results are obtained only under the assumption that trees have bounded depth [2].

For tree structured data specialized query languages have been proposed. The most notable example is XPath, an XML query language used widely in practice and extensively studied (see e.g., [5, 10, 13, 14]). Recently, a new formalism was introduced for the purpose of testing equivalence of ActiveXML systems [3]. In this formalism, data trees are unlabelled but queries can use data constants (equivalently, the labelling alphabet is infinite but queries can test labels for equality); queries have a restricted form of regular tree patterns (RTPQs), which are essentially regular expressions over the set of tree patterns using child and descendant axes and data equalities. Compared to XPath, these queries have richer ways of comparing data values.

In [3], the authors consider restricted data comparisons, called XPath-joins, which mimic data comparisons allowed in XPath. Under this restriction, they show that containment of (unary) RTPQs over documents satisfying a Boolean combination of (Boolean) RTPQs can be tested in EXPTIME. This result essentially relies on EXPTIME-completeness of satisfiability for RegXPath(↓, =), i.e., XPath with child axis and data equality, extended with Kleene star operator [10]. The complexity of problems like containment, equivalence or satisfiability for general RTPQs remains open. Answers to these questions would give immediate corollaries about equivalence testing for Active XML systems. The expressive power of RTPQs in comparison to more standard formalisms is also unknown.

Our contribution. In this paper, we answer most of these questions. We consider a more uniform setting, in which RTPQs can use not only child and descendant, but also parent and ancestor axes. These generalized RTPQs turn out to be equivalent to linear monadic datalog with child, descendent, and data equality. We investigate satisfiability, containment, and satisfiability of Boolean combinations of RTPQs, both for data words and data trees. We establish complexity of these problems for several combinations of axes allowed in the queries. Most importantly, we show that

- over unlabelled data words containment is undecidable in general, but becomes PSPACE-complete if only short axes are allowed;
- over unlabelled data trees containment is undecidable in general, but becomes EXPSPACE-complete if queries use only forward axes.

The second result solves the open problem from [3]. Using the equivalence with linear monadic datalog, we strengthen the result from [2] by showing that containment of linear monadic datalog programs in unions of conjunctive queries is undecidable even over unlabelled data trees or data words.

Organization. In Section 2 we introduce notation and definitions used throughout the paper. In Section 3 we explore the connection between RTPQs and datalog. The word case is investigated in Section 4, and the tree case in Section 5. We conclude the paper with possible directions for future research.

2. PRELIMINARIES

We consider finite trees labelled with symbols from an alphabet A. We are mainly interested in infinite alphabets. We use standard notation for child, descendant, parent and ancestor relations ($\downarrow, \downarrow_+, \uparrow, \uparrow^+$). The set of all these relations is denoted by τ_{bin}.

Let $\Delta = \{x_1, x_2, \dots\}$ be an infinite set of variables and let $\tau \subseteq \tau_{bin}$ be a non-empty set of relations. A tree pattern query (TPQ) over τ is a tree labelled with elements of $A \cup \Delta$ with edges from τ. Nodes of π labelled with elements of A are called label nodes, and nodes with Δ labels are variable nodes. Additionally, each TPQ π has two distinguished nodes: in and out, denoted by $in(\pi)$ and $out(\pi)$. Throughout the paper we assume that $in(\pi)$ is always the root of π. The set of all TPQs over τ is denoted by $\mathsf{TPQ}(\tau)$.

Let t be a tree over A, $nodes_t$ denotes the set of nodes of t and $lab_t : nodes_t \to A$ is the labelling of t. We use analogous notation for TPQs.

DEFINITION 1 (HOMOMORPHISM). *Let $\pi \in \mathsf{TPQ}(\tau)$ and let t be a tree over the set A. We say that*

$$h : nodes_\pi \to nodes_t$$

is a **homomorphism** *if*

- *$lab_t(h(v)) = lab_\pi(v)$ for every label node v;*
- *$lab_\pi(v) = lab_\pi(w)$ implies $lab_t(h(v)) = lab_t(h(w))$ for all variable nodes v, w; and*
- *h preserves binary relations from τ.*

We say that h is a **rooted homomorphism** if the root of π is mapped to the root of t. We write $t \models \pi$ if there is a rooted homomorphism from π to t.

Let $\Pi = \pi_1 \cdot \ldots \cdot \pi_n$ be a word over the alphabet $\mathsf{TPQ}(\tau)$, with $\pi_i \in \mathsf{TPQ}(\tau)$. We say that $h : \Pi \to t$ is a homomorphism iff there are homomorphisms h_i for every $i = 1, \dots, n$ such that:

- each h_i is a homomorphism from π_i to t;
- $h_i(out(\pi_i)) = h_{i+1}(in(\pi_{i+1}))$ for every $i < n$.

If the root of π_1 is mapped to root of t then we say that h is a rooted homomorphism and write $t \models \Pi$.

DEFINITION 2 (RTPQ). *A* **regular tree pattern query** *(RTPQ) φ over τ is a regular expression that uses elements of $\mathsf{TPQ}(\tau)$ as letters. By $L(\varphi)$ we denote the language of words accepted by φ. We write $t \models \varphi$ iff there is a word $\Pi \in L(\varphi)$ such that $t \models \Pi$.*

The set of all RTPQs over τ is denoted $\mathsf{RTPQ}(\tau)$. The multiset of all TPQs used in RTPQ φ is denoted $P(\varphi)$. For example, if π is a TPQ and $\varphi = \pi^* \cdot \pi$ then $P(\varphi)$ has two elements. We use notation $L_{pref}(\varphi)$, $L_{inf}(\varphi)$, and $L_{suf}(\varphi)$ for the sets of prefixes, infixes, and suffixes of words from $L(\psi)$. Since we distinguish different occurrences of identical TPQs in $P(\varphi)$, we can determine whether a word $w \in L_{inf}(\varphi)$ belongs to $L_{pref}(\varphi)$ or $L_{suf}(\varphi)$ by looking at the first and last letter of w. Let $P_{first}(\varphi)$ and $P_{last}(\varphi)$ denote, respectively, the sets first and last letters of words in $L(\varphi)$.

We will also study patterns that are words and have words as models. For them we use analogous notation changing the letter T to W: we talk about WPQs, RWPQs, $\mathsf{WPQ}(\tau)$ etc. For such queries, instead of $\uparrow$ and $\downarrow$, we use $\rightarrow$ and $\leftarrow$, respectively; we also often skip the symbol $\rightarrow$ to enhance readability.

The following example shows the power of RWPQs. We shall construct a counter, enumerating all values between 0 and $2^n - 1$. The RWPQ will have size $\mathcal{O}(n)$. We will use only two symbols from the alphabet to implement the counter. The counter value will be stored in binary in n symbols. A crucial observation is that to increase correctly the counter, we need to find the least significant position with 0. Then we can change this position to 1 and all less significant bits to 0 (because there were only 1's on less significant positions). All positions to the left to the first 0 remain unchanged. To increase the counter when the least significant 0 is on the ith position, we use pattern inc_i:

$$inc_i = (X_1)^{in} X_2 \cdots X_{n-i} 01 \cdots 1 \rightarrow \\ \rightarrow (X_1)_{out} X_2 \cdots X_{n-i} 10 \cdots 0\,.$$

Then, we combine inc_i patterns into a counter going from 0 to desired value ($2^n - 1$ in this case):

$$c_n = val_0 \cdot \left(\bigcup_{1 \le i \le n} inc_i \right)^* \cdot val_{2^n - 1} \qquad (1)$$

where val_k is number k stored in n bits in binary, e.g.,

$$val_0 = (0)^{in}_{out} 0 \cdots 0$$

If a word satisfies c_n, then it must contain all values from 0 to $2^n - 1$. Note that by locating the out node in inc_i on the $(n+1)$-th position, we pass many values between two consecutive WPQs. We shall use this feature often.

We shall investigate three classical problems of static analysis: satisfiability (SAT), containment (CON) and satisfiability of Boolean combinations (BC-SAT) of queries. In our setting, BC-SAT for trees is less interesting than containment: RTPQs are closed under disjunction and conjunction (disjunction is built-in, conjunction can be achieved by merging the roots of the RTPQs), therefore each Boolean combination of RTPQs can be expressed in the form $\varphi \wedge \neg\psi$ where φ and ψ are RTPQs. Merging roots of many RTPQs can result in exponential blow-up: if we merge n queries, each a disjunction of two TPQs, the resulting query may have to be written as a disjunction of 2^n possible TPQs. In principle this could affect the complexity but in fact, as we shall see, BC-SAT and CON have the same complexities in all the cases we consider.

3. DATALOG AND RTPQS

In this short section we establish the connection between RTPQs and datalog. First, we briefly recall the syntax and semantics of datalog. For formal semantics or a broader background we refer the reader to [8] or [1].

DEFINITION 3 (DATALOG). *A* **Datalog program** $\mathcal{P}$ *is a set of rules of the form*

$$head \leftarrow body\,,$$

where head is an atom and body is a (possibly empty) conjunction of atoms written as a comma-separated list. There is one designated rule called the **goal** *of the program. The atoms in the body use predicates from the underlying structure or predicates defined in $\mathcal{P}$; the predicate used in the head cannot be used in the underlying structure. Each variable in the head must be used in the body, and if there are variables in the body that are not in the head, we assume they are quantified existentially. The program is evaluated by generating all atoms that can be inferred from the underlying structure by applying the rules repeatedly, to the point of saturation, and then taking atoms matching the head of the goal rule.*

The predicates defined in the program are called **intensional**, and the predicates of the underlying structure are called **extensional**.

The following program describes property "there is a path from the root of the tree to a leaf labelled only with a's":

$$\begin{aligned} \underline{\mathrm{G}} &\leftarrow P(X), root(X). \\ P(X) &\leftarrow a(X), child(X,Y), P(Y) \\ P(X) &\leftarrow a(X), leaf(X) \end{aligned}$$

The predicate G is the goal of the program, P and G are intensional predicates and $root$, $leaf$, a, and $child$ are extensional ones.

A Datalog program is **linear** when the right-hand side of every rule contains at most one atom with intensional predicate, and **monadic** when each intensional predicate is at most unary. For instance, in the program

$$\begin{aligned} R(x,y) &\leftarrow\ P(y) \\ P(x) &\leftarrow\ child(x,y), child(x,z), P(y), P(z) \\ B(x) &\leftarrow\ C(x) \end{aligned}$$

The first rule is linear but not monadic, the second rule is monadic ($child$ is extensional) but not linear, and the last rule is linear and monadic.

We can now show the connection between the two formalisms.

PROPOSITION 1. $\mathsf{RTPQ}(\downarrow, \downarrow_+, \uparrow, \uparrow^+)$ *and linear monadic datalog have the same expressive power.*

PROOF. We shall show that for every monadic, linear datalog program P, there is equivalent RTPQ φ, and conversely, for every RTPQ φ there is equivalent linear nad monadic program P.

From RTPQs to datalog. The conversion from an RTPQ to a datalog program is straightforward. Let φ be an RTPQ. Each pattern π used in φ can be rewritten as a datalog rule $R_\pi(x) \leftarrow body_\pi$. Each node of the pattern becomes one variable in the program and appropriate relations are copied. For example, pattern $\pi = (X)^{in} \downarrow (1)_{out}$ gives rule $R_\pi(x) \leftarrow child(x, z), 1(z)$, where the node with label X became variable x and the node with label 1 became z. Without loss of generality we can assume that the variable for the in node of π (the variable in the head) is always x and variable for the out node is always z (or x in case the in and out nodes are the same node).

To create a program P_φ equivalent to φ, we first turn φ to a finite automaton $\mathcal{A}_\varphi$ over the alphabet consisting of TPQs occurring in φ. Then, the program P_φ is created as follows. For each state q of $\mathcal{A}_\varphi$, we introduce a new intensional predicate $q(x)$, and for each transition (q, π, q') in $\mathcal{A}_\varphi$ we add a rule $q(x) \leftarrow body_\pi, q'(z)$ (or $q(x) \leftarrow body_\pi, q'(x)$ if the in and out nodes of π coincide). Additionally, if q is accepting, we add rule $q(x) \leftarrow$. The goal of P_φ is $G \leftarrow root(x), q_0(x)$, where q_0 is the initial state of $\mathcal{A}_\varphi$.

From datalog to RTPQs. Let P be a linear monadic datalog program. First, for each rule r in P, we construct graph describing this rule. Variables become nodes and axes become edges (there are four edge types: $\downarrow, \downarrow^+, \uparrow, \uparrow^+$). Unary relations turn variables into constants, e.g., $1(X)$ in a rule causes all nodes with X to be relabelled to 1. Data equality unifies variables. This graph will be a pattern in the RTPQ. If this graph contains a cycle, then no tree can satisfy such rule, so we can restrict our attention to DAGs. Finally, we can turn each such DAG into a disjunction of TPQs, using the standard construction illustrated in Figure 1.

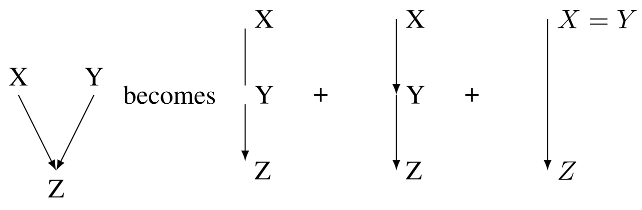

Figure 1: Turning DAG patterns to disjunctions of TPQs

The in node is the node corresponding to variable in the head of the rule and the out node is the node corresponding to the variable used by the only intensional predicate in the body of the rule (there can be at most one such predicate because P is linear). If there is no intensional predicate in the rule's body, then the out node can be any node as this pattern will not be connected to any further patterns. Let $\mathcal{T}$ be the set of all tree patterns obtained by transforming the rules of P.

We now construct a finite state automaton that uses $\mathcal{T}$ as its alphabet. States of $\mathcal{A}$ will be $\top, \bot$ and intensional predicates of P. Transitions of $\mathcal{A}$ will agree with rules of P. Let π be a TPQ obtained from rule $R(x) \rightarrow \ldots, Q(z)$, where Q is intensional predicate. We add transition (R, π, Q) to $\mathcal{A}$. If there is no Q, we use $\top$ instead. $\top$ is the only accepting state. Initial state is the goal of P. Thus we obtain a finite state automaton that recognises language L over $\mathcal{T}$. We can easily construct regular expression defining this language. This expression is an RTPQ and trees satisfying this RTPQ are exactly those accepted by P. $\Box$

In [2] the authors present also a definition of conjunctive queries (CQ in short). A conjunctive query is an existential formula of the form $\exists x_1 \ldots x_k \varphi$. Sometimes we will consider unions of conjunctive queries (UCQs), i.e., queries that do not use recursion.

4. WORDS

We begin our analysis from words, because some tedious technical difficulties can be avoided in this case. The main undecidability result for trees is a direct consequence of the undecidability of RWPQ($\rightarrow, \rightarrow^+$).

We start with simplest case: patterns using only $\rightarrow$ or $\rightarrow, \leftarrow$ relations. Even these simplest RWPQs are not trivial, as satisfiability, containment and BC-SAT are PSPACE-complete for them.

THEOREM 1. *Satisfiability, containment and BC-SAT for* RWPQ($\rightarrow$) *and* RWPQ($\rightarrow, \leftarrow$) *are* PSPACE*-complete.*

We prove this theorem by showing that

1. satisfiability for RWPQ($\rightarrow$) is PSPACE-hard, which implies hardness of all three problems for RWPQ($\rightarrow$) and RWPQ($\rightarrow, \leftarrow$); and
2. BC-SAT for RWPQ($\rightarrow, \leftarrow$) is in PSPACE.

We prove the hardness first.

LEMMA 1. *Satisfiability for* RWPQ($\rightarrow$) *is* PSPACE*-hard.*

PROOF. To prove hardness, for a number n and a Turing machine M, we construct RWPQ of size polynomial in $|M|$ and n that is satisfiable iff M accepts the empty word using not more than n tape cells.

Assume that Σ is the tape alphabet of M, Q is the set of states, F is the set of accepting states and δ be its transition relation. The alphabet A used by our RWPQ will be any infinite set containing Q and two copies of Σ, denoted Σ and $\widehat{\Sigma}$ (symbols from $\widehat{\Sigma}$ are denoted with $\widehat{\ }$). The symbols from $\widehat{\Sigma}$ will be used to mark the position of the head on the tape, so $\widehat{a}$ means that the machine's head is over symbol a.

The basic building block of our RWPQ will be a pattern for one transition rule and position on the tape

$$(q_1)^{in} X_1 X_2 \ldots Head \ldots X_n \rightarrow \tag{2}$$

$$\rightarrow (q_2)_{out} X_1 X_2 \ldots Head' \ldots X_n \tag{3}$$

where q_1, q_2 are states of M, $Head$ and $Head'$ are 3-symbol descriptions of the tape contents around the machine's head.

Specific content of $Head$ will be determined by the type of transition (whether the head moves right, left or stays in the same position). Note that the out node is located on the state symbol in the second configuration. This way, when we apply Kleene star to our pattern, two consecutive patterns will overlap, ensuring the integrity of the run.

For different head movements, patterns will be different but very similar. For example, transition rule $\delta \ni d =$ "from state q_1, seeing letter a, change state to q_2, write c, and move the head to the right" with head on i-th position and letter b on the next position of the head, we use the following pattern $\varphi_{d,i,b}$:

$$\begin{aligned}\varphi_{d,i,b} &= (q_1)^{in} X_1 X_2 \dots X_{i-1} \widehat{a}\, b X_{i+2} \dots X_n \\ &\to (q_2)_{out} X_1 \dots X_{i-1} c\, \widehat{b} X_{i+2} \dots X_n\end{aligned}$$

The variables X_i repeating in both configurations ensure that the tape contents other than symbol under the head will not change. Symbol $\widehat{a}$ is changed to c and b becomes $\widehat{b}$ to mark the new position of the head. Additionally, there must be separate patterns for head positions on first and last tape symbol (but only for rules that will keep the head within the tape; in other words, we disallow moves left from position 1 and right from position n). Patterns for the head moving left and staying in the same position are very similar.

The final RWPQ will be

$$\varphi_0 \circ \left(\bigcup_{(d,i,b) \in \delta \times [1 \dots n] \times \Sigma} \varphi_{d,i,b} \right)^* \circ \varphi_F$$

where

$$\begin{aligned}\varphi_0 &= (q_0)^{in}_{out} \widehat{\bot} \bot \cdots \bot \\ \varphi_F &= \bigcup_{q \in F} (q)^{in}_{out} X_1 X_2 \dots X_n\end{aligned}$$

describe initial and final configurations, respectively.

If there is a word matching this RWPQ then it describes a correct and accepting run of M. Obviously, such run uses at most n tape cells. On the other hand, if there is an accepting run that does not exceed n cells, then there is coding of it into a word, and our RWPQ will be satisfiable.

One remaining issue is the size of constructed query. Initial and final patterns have size $n + 1$ and the transition patterns have size $2(n + 1)$ each. There are at most $n|\delta||\Sigma|$ patterns in the disjunction, so the total size of the query is polynomial in $|M|$ and n. □

Before we give the PSPACE algorithm for BC-SAT, we prove an auxiliary lemma.

LEMMA 2. *Let φ be a boolean combination of $\varphi_1, \dots, \varphi_k \in$ RWPQ($\to, \leftarrow$) over an infinite alphabet A. There exists a finite alphabet $A_\varphi \subseteq A$ such that φ is satisfiable over A iff φ is satisfiable over A_φ. Moreover, the size of A_φ is linear in size of φ.*

PROOF. Let B be the set of the letters that appear in φ as labels, let n be the length of the longest WPQ used in φ and let $C \subseteq A$ be an alphabet with n letters such that $B \cap C = \emptyset$. We define $A_\varphi = B \cup C$. Set $|A_\varphi| = m$.

The right to left implication is obvious. Arguing in the other direction, suppose we have a word v over A that satisfies φ. If it uses not more than m letters then we are done. Otherwise we proceed as follows.

Our goal is to modify the word v into a word over the alphabet A_φ. The procedure reads consecutive letters of the word v and puts some of them on a list l. We denote the size of the list $|l|$. Let i be the position of the last read letter in v, and let a be the last letter on the list l. If the letter v_i is from the alphabet B we do nothing. If v_i is already on the list we move this letter to the beginning of the list. Assume $v_i \notin B$ and $v_i \notin l$. If $|l| < n$, we add v_i at the beginning of l. If $|l| = n$, we modify the word v as follows: (1) for all $j > i$, if $v_j = a$ (the last letter in l), we change v_j to a new letter that does not occur in v, (2) we replace all occurrences of the letter v_i in v with the letter a, and (3) we move the letter a to the beginning of the list. We continue this procedure until we reach the end of the word. In the end we obtain a word which has at most m letters. We denote this word by u.

We now prove that word u satisfies φ. It suffices to show that $u \models \varphi_k \iff v \models \varphi_k$ for all k. Take $\pi \in P(\varphi_k)$. Since the variables in queries are local, it is enough to prove that ($\heartsuit$) $u[i,j] \models \pi \iff v[i,j] \models \pi$ for all i, j. Since $|\pi| \leq n$, we may assume that $j - i \leq n$. Under this assumption it is easy to verify that the procedure ensures that for all $i \leq i_1, i_2 \leq j$ it holds that $v_{i_1} = v_{i_2} \iff u_{i_1} = u_{i_2}$, which proves ($\heartsuit$). □

LEMMA 3. *BC-SAT for* RWPQ($\to, \leftarrow$) *is in* PSPACE.

PROOF. Recall that $L(\psi)$ is the language of all words over the set of WPQs generated by ψ, and $L_{pref}(\psi)$, $L_{inf}(\psi)$, $L_{suf}(\psi)$ stand for the language of prefixes, infixes, and suffixes of words from $L(\psi)$. The sets $P_{first}(\psi)$ and $P_{last}(\psi)$ are respectively the set of all WPQs that are last letters and first letters in words from $L(\psi)$.

Let φ be a boolean combination of queries $\varphi_1, \dots, \varphi_k \in$ RWPQ($\to, \leftarrow$) over an infinite alphabet A. By Lemma 2, the satisfiability of φ can be verified over a finite alphabet A_φ. Let n be the length of the longest WPQ used in ψ.

For each i we construct a deterministic finite automaton $\mathcal{A}_i$ which recognises words over the alphabet A_φ that satisfy φ_i. We denote by w the whole word read by $\mathcal{A}_i$ and by w_s its suffix with last n letters. Each state of $\mathcal{A}_i$ has two components.

The first component is a word from $\bigcup_{k \leq n} (A_\varphi)^k$, corresponding to the suffix w_s (initially, there are less than n letters to remember).

The second component is a subset of

$$[n+1] \times P(\varphi_i) \times [n+1] \times P(\varphi_i)\,,$$

where $[k] = \{1, 2, \dots, k\}$. For $p, p' \leq n$, each element (p, π, p', π') of the second component corresponds to a subquery $\psi \in L_{inf}(\varphi_i)$, whose first WPQ is π and last WPQ is π', and a homomorphism from ψ to w that maps $in(\pi)$ to $w[|w| - |w_s| + p]$ and $out(\pi')$ to $w[|w| - |w_s| + p']$. The intended meaning is that for all homomorphisms from queries in $L_{inf}(\varphi_i)$ to w, the state q remembers the information about the first and last node. This information is sufficient to build information about bigger queries by induction because the variables are local.

Note that there is an additional value $n + 1$ available on the first and third coordinate. In the first coordinate, value $n+1$ is used to indicate that the represented homomorphism is a rooted homomorphism from a subquery $\psi \in L_{pref}(\varphi_i)$ to w, and we only remember the information about the last WPQ of ψ. In the third component, value $n + 1$ is used to indicate that $\psi \in L_{suf}(\varphi_i)$, and the query does not have to be extended any more. If a state contains a tuple with the first and third entry equal to $n + 1$, it means that the represented ψ belongs to $L(\varphi_i)$ and the represented homomorphism is rooted. Such states are accepting.

We now explain how the transition function δ_i works. The automaton starts in state $(\varepsilon, \emptyset)$. If $\mathcal{A}_i$ is in state q_1 and reads a new letter a, it moves to state q_2, whose first component contains updated suffix of length at most n and second component of q_2 is a set Δ defined as follows.

Set Δ contains all elements of the second component of q_1, but the first and third coordinates are decreased by 1 (unless the remembered prefix is still shorter then n). If they drop below 1, q_2 forgets about them.

Moreover, for all $\pi \in W(\varphi)$, if there is a homomorphism η from π to w_s and the image contains the new letter then Δ contains all elements $(\eta(in(\pi)), \pi, \eta(out(\pi)), \pi)$. If $\pi \in P_{last}(\varphi_i)$, we also add tuple $(\eta(in(\pi)), \pi, n + 1, \pi)$. If the remembered suffix has length strictly smaller than n, $\eta(in(\pi)) = 1$, and $\pi \in P_{first}(\varphi_i)$, the we also add tuple $(n + 1, \pi, \eta(out(\pi)), \pi)$. If both of these hold, we add tuple $(n + 1, \pi, n + 1, \pi)$.

Finally, we close Δ under legal concatenations: if Δ contains tuples $(p_1, \pi_1, p_2, \pi_2), (p_3, \pi_3, p_4, \pi_4)$ such that $p_2 = p_3$ and $\pi_2 \cdot \pi_3 \in L_{inf}(\varphi_i)$, then Δ should also contain tuple (p_1, π_1, p_4, π_4). (Here we rely on the fact that occurrences of identical WPQs in φ_i are treated as different WPQs.)

Since the constructed automata $\mathcal{A}_i$ are deterministic, the standard product construction gives an automaton equivalent to φ. The size of $\mathcal{A}_i$ is exponential in size of φ, but states and transitions of $\mathcal{A}_i$ can be generated in polynomial space. To check its emptiness we use the reachability problem which is in NLogSpace. Altogether, this gives us a PSpace algorithm. □

We now extend the set of available axes with $\rightarrow^+$. First, we present main result of this section – undecidability of containment for RWPQ$(\rightarrow, \rightarrow^+)$. In fact, we prove a stronger result.

Theorem 2. *Containment of* RWPQ$(\rightarrow, \rightarrow^+)$ *queries in UCQs using* $\rightarrow, \rightarrow^+$ *and label comparisons is undecidable.*

Proof. To prove undecidability, we reduce from the following tiling problem: given

- a set of tiles K,
- a horizontal correctness relation $H \subseteq K \times K$,
- a vertical correctness relation $V \subseteq K \times K$,
- an initial tile k_0,
- a final tile k_F,

decide if it is possible to create a finite tiling on the grid that starts with k_0, has k_F as the last tile and all pairs of adjacent tiles satisfy appropriate relation, H or V.

We encode the tiling as a word over $\mathbb{D} = K \cup \mathbb{N} \cup \{\#\}$. Placing a tile is represented by four letters, (n, n', m, t), where $n, n', m \in \mathbb{N}$ denote row number, next row number, and column number, and $t \in K$ is the tile placed in position (n, m) in the grid. The tiling is encoded row by row in the word. Consecutive rows are separated by $\#$'s.

We shall construct two queries φ and ψ. Query will φ describe constraints enforced by tiling relations, initial configuration and ending tile. Query ψ will be a union of conjunctive queries and $\neg\psi$ will be used to guarantee that matched word is a proper encoding of some tiling.

To guarantee that horizontal and vertical tiling relations are satisfied we use query α:

$$\begin{aligned}\alpha = \bigvee_{(\tau,\tau')\in H, (\tau,\tau'')\in V} & (R)^{in} \rightarrow R' \rightarrow C \rightarrow \tau \rightarrow \\ & (R)_{out} \rightarrow R' \rightarrow C' \rightarrow \tau' \rightarrow^+ \\ & R' \rightarrow R'' \rightarrow C \rightarrow \tau'' \rightarrow \\ & R' \rightarrow R'' \rightarrow C'\end{aligned}$$

where R, R', R'', C, C' are variables, R, R', R'' represent row numbers, and C, C' represent column numbers. Last three nodes of α enforce that the tiles form a proper grid, without it, there could be some unnecessary tiles between column C and C' in the row R'.

In the last row of the grid, we do not need the vertical constraints, so the following simpler formula will suffice:

$$\begin{aligned}\beta = \bigvee_{(\tau,\tau')\in H} & (R)^{in} \rightarrow R' \rightarrow C \rightarrow \tau \rightarrow \\ & (R)_{out} \rightarrow R' \rightarrow C' \rightarrow \tau'\end{aligned}$$

The positive query φ is defined as

$$\varphi_0 \cdot (\varphi_{join})^* \cdot \varphi_F \tag{4}$$

where

$$\begin{aligned}\varphi_0 = & (\#)^{in}_{out} \rightarrow 1 \rightarrow 2 \rightarrow 1 \rightarrow k_0 \\ & \rightarrow^+ 1 \rightarrow 2 \rightarrow C \rightarrow X \rightarrow \# \\ & \rightarrow^+ Y \rightarrow Z \rightarrow C \rightarrow k_F\end{aligned}$$

enforces the initial tile and ensures that the final tile k_F is in the last column that is present in the first row. Because of the

RWPQs properties, each row can be longer than the previous one, so even if k_F is in the last row, this does not guarantee that the tiling is correct. Last row of φ_0 solves this issue.

$$\begin{aligned}\varphi_F =&((\#)^{in} \to (R)_{out}) \cdot \beta^* \cdot \\ &\cdot ((R)^{in} \to R' \to C \to k_F \to (\#)_{out})\end{aligned}$$

enforces correct last row and the last tile of the tiling, and

$$\begin{aligned}\varphi_{join} =&((\#)^{in} \to (R)_{out}) \cdot \alpha^* \cdot \\ &\cdot ((R)^{in} \to R' \to C \to Q \to \\ &(\#)_{out} \to R' \to R'' \to 1 \to Q')\end{aligned}$$

ensures correctness of the all the remaining rows.

Each time α is used, it enforces that consecutive tiles match the horizontal relation H and that for each tile t_i there exists another tile with correct row and column that matches the vertical relation V. Similarly for β. Note that apart from enforcing tiling constraints, φ also guarantees that each word matching φ has only symbols form K on positions corresponding to tiles and that between two $\#$'s the row and next row numbers are the same.

The negative query ψ will be a disjunction of queries describing possible errors in the encoding. This way, each word satisfying $\neg\psi$ will be a correct encoding of the tiling. Possible coding errors are:

1. $\#$ or tile symbol appearing in wrong position,
2. two consecutive $\#$ symbols,
3. the same row number used in two different rows,
4. the same column number twice in one row.

The positive query, φ, ensures that row and next row are the same between two $\#$'s, so to find $\#$ or tiles used as row numbers it is enough to check first and second position after each $\#$. For example

$$\exists x_1, x_2, x_3 \#(x_1) \wedge x_1 \to x_2 \to x_3 \wedge t(x_3)$$

checks if tile t appears as a next row identifier. To find tiles or $\#$'s used as column identifiers, we will rely on the fact that column number always precedes a tile:

$$\exists x_1, x_2 \quad t_1(x_1) \wedge t_2(x_2)$$

Disjunction of such queries for all elements of $(K \cup \{\#\}) \times K$ will find wrong symbols used as column numbers. Query for two consecutive $\#$ symbols is:

$$\exists x_1, x_2 \quad \#(x_1) \wedge \#(x_2) \wedge (x_1 \to x_2)$$

Other errors are equally easy to define.

Observe that each word satisfying $\varphi \wedge \neg\psi$ contains correct tiling of the grid (only contains because each row can be longer than the previous one), therefore $\varphi \wedge \neg\psi$ is satisfiable iff there exists solution to the tiling problem. □

To complete the picture for RWPQ$(\to, \to^+)$, we establish the complexity of satisfiability.

THEOREM 3. *Satisfiability problem for* RWPQ$(\to, \to^+)$ *is* NEXPTIME*-complete.*

PROOF. Let us begin with the upper bound. Let $\varphi \in$ RWPQ$(\to, \to^+)$ over alphabet A and let $\Sigma \subseteq A$ be the set of labels appearing in φ. Observe that if some w satisfies φ then, unless Σ is empty, there exists $w \in \Sigma^*$ that satisfies φ: since φ does not use any form of negation, we can replace each letter not in Σ with an arbitrary letter from Σ. We will assume that Σ is not empty because if Σ is empty then our model w can be built from one letter. We will show that RWPQ$(\to, \to^+)$ has exponential model property: if φ is satisfiable, then it is satisfied in some word of length single exponential in the size of φ.

Suppose that φ is satisfied in a word w and consider a homomorphism witnessing this fact. Let π be a pattern from φ (recall that different occurrences of the same pattern in φ count as two separate patterns). Consider all the subwords where π is matched. If any two such subwords are the same then we can merge those two subwords and remove every symbol between those two subwords and the resulting word will still satisfy φ, e.g., if $\varphi = ((X)^{in} \to (Y)_{out})^*$, $\pi = (X)^{in} \to (Y)_{out}$, $w = aabcdaa$, we can remove bcd, merge aa's in w, and the obtained word aa also satisfies φ. Let w' be the shortened word. In w', each pattern π is matched in at most $\Sigma^{|\pi|}$ different positions. Cutting off the suffix of w' that is not reached by query φ, we obtain a model of size bounded by $|\varphi| \cdot |\Sigma|^{|\varphi|}$, because there are at most $|\varphi|$ patterns in φ and each pattern has length at most $|\varphi|$.

To check satisfiability of φ it suffices to guess a word $w \in \Sigma^*$ of length $|\varphi| \cdot |\Sigma|^{|\varphi|}$ and a homomorphism from φ to w, and verify that it is indeed a homomorphism. All this can be done in NEXPTIME.

To prove hardness, we will reduce the exponential tiling problem. This is the same problem as described on page 6, except that a natural number n is given additionally and the goal is to fill the $2^n \times 2^n$ grid. To obtain the reduction for the exponential tiling problem, we modify the reduction from the unrestricted tiling problem, given in the proof of Theorem 2, and the binary counter query c_n described on page 3. The most important part of the tiling reduction was the query α, which ensured that all adjacent tiles respect the relations H and V. The query was a disjunction of patterns

$$RR'C\tau \to RR'C'\tau' \to^+ R'R''C\tau'' \to R'R''C' ,$$

where R, R', R'' represent rows, C, C' represent columns, and $(\tau, \tau') \in H$, $(\tau, \tau'') \in V$. In the proof of Theorem 2, we used a negated UCQ to ensure that the row R and the next row R' are consistent throughout the encoding. This time, as there is no negation available, we will use binary counters to store the row and column number. This means that we will be able to address only exponentially many rows and columns, which is just right for our purpose. Each node of the grid is thus encoded by $2n + 1$ symbols: n for the row number, n for the column number, and 1 for the tile. The queries

$$(X_1)^{in} X_2 \cdots X_{n-i} \underbrace{01\cdots 1}_{i} \to Y_1 \cdots Y_{n-j} \underbrace{01\cdots 1}_{j} \tau \to$$

$$(X_1)_{out} X_2 \cdots X_{n-i} \underbrace{01\cdots 1}_{i} \to Y_1 \cdots Y_{n-j} \underbrace{10\cdots 0}_{j} \tau' \to^+$$

$$X_1 \cdots X_{n-i} \underbrace{10\cdots 0}_{i} \to Y_1 \cdots Y_{n-j} \underbrace{01\cdots 1}_{j} \tau'',$$

with $(\tau,\tau') \in H, (\tau,\tau'') \in V$ and $1 \le i,j \le n$, describe possible tiles adjacent horizontally and vertically in the grid. The construction of the whole query is analogous to the one described in the proof of Theorem 2. □

As we can see, allowing $\to^+$ greatly increases the complexity of satisfiability and makes containment (and BC-SAT) undecidable.

5. TREES

Let us start with the simple cases.

THEOREM 4. *For* RTPQ($\downarrow$), *containment and BC-SAT are* PSPACE-*complete.*

We show how to prove this after showing the proof for Theorem 7.

THEOREM 5. *For* RTPQ($\downarrow, \uparrow$), *containment and BC-SAT are* EXPTIME-*complete.*

PROOF. The proof of this theorem is very similar to the proof of Theorem 1. We construct a deterministic bottom-up automaton exponential in the size of φ. Like in the proof of Theorem 1, states remember information about satisfiable subwords from $L(\varphi)$. The emptiness problem for tree automata is in PTIME so this gives an EXPTIME algorithm.

To prove hardness we encode an emptiness of an alternating Turing machine working in polynomial space. Without loss of generality we may assume that each universal configuration has at most two successors.

We use queries similar to those in the proof of Theorem 1:

$$\varphi_{d,i,b} = M q_1 X_1 X_2 \ldots X_{i-1} \widehat{a}\, b X_{i+2} \ldots X_n$$
$$\downarrow M' q_2 X_1 \ldots X_{i-1} c\, \widehat{b} X_{i+2} \ldots X_n$$

Note that before configurations we now use a new symbol $M, M' \in \{T, N, L, R\}$. If q_1 is universal, the first configuration in the queries $\varphi_{d,i,b}$ has two following configurations, one starting with L and the other with R. Configurations after existential states start with N and in the root we have T. The *in* nodes are in the roots. The *out* nodes are in the nodes labeled with labels from $\{T, N, L, R\}$. If the state was universal then the *out* node is the one labeled with L. We make analogous changes for φ_0 and φ_F from the proof of Theorem 1.

Let φ_p be the query resulting by carrying out the recipe from Theorem 1. This query checks if the leftmost branch of the computation is correct. Using $\uparrow$ we can build a query that moves upwards looking at the configurations. When it sees R or N it can move further upwards. If it sees L then it moves to the configuration with R. This can be done with queries $\varphi_{d,i,b}$ with universal states in the first configuration, but with the *in* node moved to L and the *out* node moved to R. Then it can proceed as before. The query is satisfied when it reaches the root labeled with T, i.e., when it has checked all runs.

Recall the queries φ_0 and φ_F from Theorem 1. The problem is that here a node can have several children so it is easy to find a model for $\varphi_0 \cdot \varphi_F$. We use the negative query ψ to take care of this problem. The query ψ is the disjunction of queries

$$\begin{tikzcd}[row sep=small, column sep=tiny]
& z_1 \arrow[d] & \\
& z_2 \arrow[dl] \arrow[dr, "+"'] & \\
M \arrow[d] & & M \arrow[d] \\
x_1 \arrow[d] & & y_1 \arrow[d] \\
\cdots \arrow[d] & & \cdots \arrow[d] \\
x_i \arrow[d] & & y_i \arrow[d] \\
k \arrow[d] & & k' \arrow[d] \\
x_{i+1} \arrow[d] & & y_{i+1} \arrow[d] \\
\cdots \arrow[d] & & \cdots \arrow[d] \\
x_n & & y_n
\end{tikzcd}$$

where x_j, y_j, z_j are variables, $i \in \{0, \ldots, n-1\}$, $M \in \{L, R, N, T\}$, and k, k' range over all possible pairs of non-equal states or letters. □

THEOREM 6. *For* RTPQ($\downarrow, \downarrow_+, \uparrow, \uparrow^+$), *containment and BC-SAT are undecidable. Moreover, already containment of RTPQs in UCQs is undecidable.*

PROOF. This is a direct consequence of theorem 2. We change the queries used in the word case by replacing $\to$ with $\uparrow$ and $\to^*$ with $\uparrow^+$, and add $X{\downarrow}\bot$ at the beginning of each query. We use $\bot$ as a starting symbol for the coded word (instead of the first #). The query first finds $\bot$ somewhere in the tree and then goes up. As there is only one path up in the tree, it behaves exactly as in the case of words. □

Together with Proposition 1, this immediately gives the following fact.

COROLLARY 1. *Containment of linear monadic datalog programs in unions of conjuctive queries is undecidable over unlabelled data trees.*

Note that we had to use $\uparrow$. The fact that there is only one path (in a word, or going up in the tree) allows the query to express much stronger properties. Indeed, if we disallow $\uparrow$ and $\uparrow^+$ axes, decidability is restored.

THEOREM 7. *Containment and BC-SAT are* EXPSPACE-*complete for* RTPQ$(\downarrow, \downarrow_+)$.

To prove this theorem we need to some auxilliary results. The first one is obvious.

LEMMA 4. *Let $\varphi \in$ RTPQ$(\downarrow, \downarrow_+)$ and let t be a tree. Assume that $v, w \in t$ satisfy $v\downarrow_+ w$ and let P be the path from v to w. Let t' be a copy of t with an additional path P' which is a copy of the path P rooted in v. Then $t \models \varphi \iff t' \models \varphi$.*

In the proof of Theorem 7 we will construct canonical models for positive queries. Roughly speaking, this model is the hardest tree for other queries to be satisfied in.

DEFINITION 4 (CANONICAL MODEL). *Let $t \models \varphi$ for some $\varphi \in$ RTPQ$(\downarrow, \downarrow_+)$. We say that t is canonical model for φ if it satisfies the following conditions:*

- *after relabelling any node of t to a fresh value, the resulting still does not model φ any more;*
- *there exists $\Pi \in L(\varphi)$ such that $t \models \Pi$ and the witnessing homomorphism $h\colon \Pi \to t$ is injective, and each node in t has a descendant in the image of h.*

A canonical model for a conjunction of RTPQs is any tree created by merging roots of canonical models for all conjuncts.

LEMMA 5. *Let $\varphi = \bigwedge_i \varphi_i \wedge \bigwedge_j \neg\varphi_j$, such that φ is satisfiable. Then there is a canonical model t for $\bigwedge_i \varphi_i$, such that $t \models \varphi$.*

PROOF. Let $t \models \varphi$, and let $\Pi_i \in L(\varphi_i)$ such that $t \models \Pi_i$. By Proposition 4 we can add to t copies of the same paths. We add them until there are such injective rooted homomorphisms $\eta_i : \Pi_i \to t$ that the images of every two homomorphisms η_i, η_j have only the root of t' in the intersection. Then we can remove every node from t that has no descendant in the image of any η_i. Finally, we change the labels in t to fresh labels in such a way that they differ whenever they can but still $t \models \Pi_i$. It is easy to see that $t \models \varphi$. $\square$

LEMMA 6. *Let*

$$\varphi = \bigwedge_{1 \le i \le n} \varphi_i \wedge \bigwedge_{1 \le j \le m} \neg\psi_j$$

be an boolean combination of RTPQs. If φ is satisfiable, then there exists a tree t such that $t \models \varphi$ and for every i and every TPQ π from φ_i and $\forall u_0\downarrow_+ v_0 \in \pi$, the length of the path between images of u_0 and v_0 in t is bounded exponentially in $|\varphi|$.

PROOF. By Lemma 5 we can assume that t is a canonical model for $\bigwedge_{1 \le i \le n} \varphi_i$. Let h be the witnessing homomorphism, let $u_0\downarrow_+ v_0$ be nodes in π, and let $u = h(u_0), v = h(v_0)$. Since t is a canonical model, each node between u and v is labelled with a unique symbol, not appearing anywhere else in the tree nor in any φ_i or ψ_j.

Let K be the product of sizes of all TPQs appearing in queries ψ_j, and suppose that length of the path between u and v is longer than K. If we cannot shorten it, it means that some (possibly many) ψ_j becomes true in this subtree after shortening. This of course means that it becomes possible to match one of these queries on the considered path (otherwise that query would be true in t before shortening). As K is greater than the size of all TPQs used in queries ψ_j, there are only two possible explanations: either some nodes $w\downarrow_+ w'$ or a subquery $(\pi')^*$ is matched at this path. The first case is not possible, because $\downarrow_+$ would not be affected by shortening. Hence, some $(\pi')^*$ is matched along our path, and since the labels our path are all unique (and π' does not contain $\downarrow_+$), π' has the form $x_1 \to x_2 \cdots \to x_l$, with all x_k's different.

Such pattern does not compare data values, it just measures l symbols on the path. Such pattern can be used only as a part of query "there exists 0 and 1 and distance between them is 4 modulo 7". This means that we have to choose such length of our path that none of these queries is matched. We know that such solution exists, because originally t satisfied φ. Consequently, there must be a solution between 0 and K, the product of sizes of all TPQs appearing in queries ψ_j. Each ψ_j contributes at most $|\psi_j|^{|\psi_j|}$ to the product, and therefore $K \le m|\varphi|^{|\varphi|}$. $\square$

PROOF OF THEOREM 7 (UPPER BOUND). We can assume without loss of generality that φ is in disjunctive normal form (as the transformation takes at worst exponential time). We guess which of the disjuncts are true. Therefore our problem is reduced to checking satisfiability of

$$\varphi = \bigwedge_{1 \le i \le k} \varphi_i \wedge \bigwedge_{k+1 \le i \le n} \neg\varphi_i$$

where $\varphi_i \in$ RTPQ$(\downarrow, \downarrow_+)$ for $i = 1, \ldots, n$.

We construct a special non-deterministic automaton $\mathcal{A}$. The goal of this automaton is to find a canonical model for φ. Essentially, $\mathcal{A}$ will be reading words of TPQ's from $L(\varphi_i)$ (right to left, or rather bottom-up, to reflect the character of axes used) and checking if any of negative queries has been matched. At the end, the automaton tries to merge the words at the root and performs one last check for matched negative patterns. If none is matched, the automaton accepts.

The alphabet of $\mathcal{A}$ is the set $\prod_{i \in \{1,\ldots,n\}}[P(\varphi_i)] \cup \{\flat\}$ where elements of $[P(\varphi)]$ are elements of $P(\varphi)$ with given lengths for every $\downarrow_+$ axis. To describe the set of states we need some definitions. Let B be the set of all non-variable labels from φ and let $\pi \in P(\varphi_i)$. Let $Pos(\pi)$ be the set of all nodes in π and all $\downarrow_+$-edges in π. Additionally, we define

$$Part(\pi) = (B \cup \{in, var, non\})^{Pos(\pi)}.$$

This set contains information about partial homomorphisms into a tree. Suppose t_ψ is a canonical model for some $\psi \in L_{suf}(\varphi_i)$ and let $f \in Part(\pi)$. Function f summarizes information about a homomorphism from π to t_ψ. The homomorphism is induced by the nodes p that satisfy $f(p) \neq non$. For our purpose it is enough to remember rooted homomorphisms from "subtrees" of π. To reflect this, we require that if $f(p), f(r) \neq non$, then all nodes on this path have value different from non. The the positions associated to $\downarrow_+$ edges are important only if they are mapped to the root of t_ψ, otherwise we ignore them. To see why we need to introduce them at all, consider a TPQ $a{\downarrow_+}b$ and suppose $t_\psi = c{\downarrow}b$. Let ψ' be a query with a root labelled with a. Then $a{\downarrow_+}b$ becomes satisfied on $t_{\varphi'\cdot\varphi}$. Thus we need to remember a partial homomorphism from $a{\downarrow_+}b$ to ψ. Since we cannot map a to the node labelled with c we need an additional position to remember, i.e., somewhere on the $\downarrow_+$ path. We interpret the other values as follows:

- if $f(p) \in B$, then p was mapped to a node labelled with one of the labels from B;
- if $f(p) = var$, then p was mapped to a node of t_ψ corresponding to a node of ψ labelled with a variable;
- if $f(p) = in$, then p was also mapped to a node of t_ψ corresponding to a node of ψ labelled with a variable, but this variable was the same as the label of the in node of the top TPQ of ψ (which we assume to be the root of ψ).

Let $B^\flat = B \cup \{\flat\}$. Automaton $\mathcal{A}$ has rejecting state q_F, accepting state q_T, and the remaining states are elements of

$$\prod_{i=1}^{k}\left(A_i \times \prod_{j=k+1}^{n} \mathcal{P}\Big(\bigcup_{\pi\in\varphi_j} Part(\pi)\Big) \times \{1,2\} \times B^\flat\right)$$

where $\mathcal{P}$ is the power set operator, and $A_i = [P(\varphi_i)] \cup \{\flat\}$. For every i, the element of A_i in the first component is the i-th coordinate of the previously read letter, and the following $n-k$ components store information about possible homomorphisms from φ_j. The last two components will be explained later. The initial state for each i has its components set to $(\flat, \emptyset, \emptyset, \ldots, \emptyset, 1, \flat)$.

We explain how the transition relation $\delta \subseteq Q \times A \times Q$ works. In the beginning the automaton works independently on every coordinate. It builds the canonical model for every φ_i for $i \in K$, ensuring that there are no homomorphisms for φ_j for $j \notin K$. In the end it glues the roots of all models for a canonical model for φ.

Fix index i. We explain the transition focusing only on the i-th coordinate of the letter it read. The word read by the automaton is a canonical model built from the bottom. Suppose π is the i-th coordinate of the read letter. The automaton checks if π can be concatenated with the previous TPQ π_{prev}. Consider the node $in(\pi)$. This node will be concatenated with the out node of the next TPQ. If the label of $in(\pi)$ is a variable then it can be changed to a label: the automaton guesses the label for this node, or that there is no label. If the labels of $in(\pi)$ and $out(\pi)$ are the same, then it copies it from the previous TPQ instead of guessing. This is kept in the last coordinate of the state in $B^\flat$. The automaton checks if the previous label guessed in $B^\flat$ matches the label of $out(\pi)$.

For all $j \geq k+1$ the algorithm looks at every TPQ $\kappa \in P(\varphi_j)$. The automaton keeps information about homomorphisms from parts of κ. For that purpose, we need to identify $f \in Part(\kappa)$ such that there is a corresponding rooted homomorphism to the canonical model built so far from the letters read by the automaton. The automaton in fact has no access to the whole canonical model, it can look only at π and the previous TPQ (stored in the first coordinate; element of A_i). But the automaton has the information about homomorphisms to the canonical model from the previous step (rooted at the previously read TPQ). This is enough to reconstruct all partial homomorphisms from κ: since the variables in TPQs are local, besides labels from B we need to know only if labels are the same or different. The automaton can find a partial rooted homomorphism to π and then check how it can be extended by looking at partial rooted homomorphisms from the previous state. We add all the identified $f \in Part(\kappa)$ to the new state, with 1 on the penultimate coordinate.

Since we are looking for homomorphisms from φ_j, we need to know if there is a word from $L(\varphi_j)$ for which the word read by the automaton is a model. That is why we need to keep information about homomorphisms from the suffixes $L_{suf}(\varphi_j)$. Let $w_S \in L_{suf}(\varphi_j)$ and let $w_S = \kappa \cdot w'_S$, where κ is a TPQ. Notice that since the in node is always in the root then we can remember the partial homomorphism from κ, if we know that there is a homomorphism from the whole word w'_S. For all $f \in Part(\kappa)$ we already added to the new state with 1 on the penultimate coordinate, we look for such words w'_S. In particular, if the TPQ κ is also in $P_s(\varphi)$, then all such functions f appear also with 2 on the penultimate coordinate. If $f(out(\kappa)) = non$, then f is not added. If the homomorphism mapped $out(\kappa)$ below the node $out(\pi)$, than f was extended from a homomorphism f' from the previous TPQ. If f' had 2 on the penultimate coordinate, then so has f. Otherwise, if $out(\kappa)$ was mapped somewhere in π, the automaton looks at the elements of $Part(\kappa)$ with 2 on the penultimate coordinate from the previous state. Since the out of π was glued to the in of the previous TPQ, the descendants of $out(\pi)$ could extend functions $g \in Part(\kappa')$ from the previous TPQ and the homomorphisms could be from the whole κ'. Moreover for some TPQs in $P(\varphi_j)$, there could be homomorphisms to π. This way the automaton can find new words w'_S (knowing only some of the first letters). If for any $\kappa \cdot w'_S \in L_{suf}(\varphi_j)$, then we also add such an f with 2 on the penultimate coordinate.

Eventually the automaton guesses a canonical tree on all coordinates $1 \leq i \leq k$. Then it merges all their roots together and checks if there are any rooted homomorphisms from any negative φ_j (for $k+1 \leq j \leq n$). If there are no such homomorphisms, then the automaton found a canonical model t; otherwise, it rejects the word.

The size of $(B \cup \{in, var, non\})^{Pos(\pi)}$ is exponential in the size of ψ for $\pi \in P(\psi)$. This gives a double exponential bound on the size of $\mathcal{P}(\sum_{\pi \in \varphi_j} Part(\pi))$ and in fact for the whole set of states Q. Since the emptiness problem is in NLOGSPACE, this means that we have an EXPSPACE algorithm. □

This proof can be modified to work on data with a finite alphabet. The functions $Part(\pi)$ would need only two values in the image, var and in, because the labels from the finite alphabet would represented in within the letters read by the automaton.

Before showing the lower bound, we prove Theorem 4.

PROOF OF THEOREM 4. The proof is a modification of the proof of Theorem 7. The automaton works in the same way, i.e., reads TPQs as letters and builds from the bottom a canonical model. Since the queries use only $\downarrow$, the automaton does not need so many states. Let s be the longest path in all TPQs from φ. The automaton can remember all TPQs that are within the distance s from the current root. We do not need to remember all of the functions $Part(\pi)$. We need to remember words $\Pi \in L_s(\varphi)$ from which there was a homomorphism. We only need to remember those homomorphisms that have roots at most s nodes below the root because here TPQs do not use the $\downarrow_+$ axis. All we need to remember about the homomorphism is: what the first letter of Π is, and where its root is mapped. This gives a bound exponential in the size of φ for the set of states. Hence, we can check the emptiness of this automaton in PSPACE (in terms of the size of φ).

To show the hardness, we can encode computations of PSPACE Turing machines, like in the case of words in Theorem 1. To avoid trivial satisfiability we use a variant of query ψ from Theorem 5. □

Finally, we shall prove that containment for $\mathsf{RTPQ}(\downarrow, \downarrow_+)$ is EXPSPACE-hard.

PROOF OF THEOREM 7 (LOWER BOUND) . This problem is equivalent to the satisfiability problem of $\varphi \wedge \neg\psi$. We will employ technique used in the proof of PSPACE-hardness in Theorem 1, but this time we use Turing machine working in EXPSPACE. Let $\mathcal{M}$ be the machine and let n be a natural number. We may assume that $\mathcal{M}$ never uses more than 2^n tape cells. The query φ will code the run of the machine, and the query ψ will ensure its correctness. This encoding is similar to the encoding of the grid in Theorem 2. Here instead of tiles we write the state and the content of the machine in its tape cells. In the proof of Theorem 2 the structure was words and we could force consistent order of identifiers for columns. On trees we cannot do it so for columns we use the binary counter.

We define a TPQ "block" as:

$$R{\downarrow}R'{\downarrow}k_1 \dots k_n{\downarrow}T{\downarrow}q$$

The R and R' variables store indentifiers of this configuration and the next configuration; on $k_1, \dots, k_n$ we have the counter; T is the content of the tape cell whose number is encoded in the counter; and in q is the state of the machine. The node with q should be labeled with an empty symbol $\flat$ for all cells except one. This way we know where the head of the machine is. By $B^i_{R,R'}$ we denote a block with i stored in the counter and R, R' as the identifiers of configurations. We use queries with such TPQs to generate the next configuration:

$$B^{i-1}_{R,R'}{\downarrow}B^{i}_{R,R'}{\downarrow}B^{i+1}_{R,R'}{\downarrow_+}B^{i-1}_{R',R''}{\downarrow}B^{i}_{R',R''}{\downarrow}B^{i+1}_{R',R''}$$

The in node will be in the R variable in $B^i_{R,R'}$ and the out node is in the R variable in $B^{i+1}_{R,R'}$.

If the i-th block has the head of the machine then we can make the machine move by one of its transitions to the next configuration, and put the head of the machine in the proper cell. If there is no head in positions $i-1, i, i+1$ then we just copy tape contents to the next configuration. Using counter increments, these TPQs can encode a run on a machine similarly to the tiling in the proof of Theorem 2.

The negative query ψ will define possible errors, which are:

- different identifiers for one configuration
- there are two different configurations with the same identifier;
- after one configuration there are two configurations with the same identifier, but different content in the cells.

All of these errors are easy to express with RTPQs. □

Satisfiability. We left the discussion of satisfiability for RTPQs for the end. Satisfiability for queries without upward axes is easily checked: the only reason why φ can be not satisfiable is when there are two concatenated queries where in node and out node have different constants. To verify that it is possible to avoid such situation, we run a reachability algorith on a graph where vertices are pairs (π, b) where π is a TPQ from φ and b is a constant that appears in φ or $\top$. Second coordinate of the pair denotes symbol enforced in the root of current pattern (or $\top$ if no symbol if enforced). Edges are only between nodes which patterns can be connected by φ. Satisfiability of $\mathsf{RTPQ}(\downarrow)$ is therefore in PTIME.

The case of RTPQs using upward axes is harder because the query can move somewhere in the bottom of the tree and then go only upwards having a word-like structure. This observation immiediately yields

- PSPACE-hardness for $\mathsf{RTPQ}(\downarrow, \uparrow)$;
- NEXPTIME-hardness for $\mathsf{RTPQ}(\downarrow, \downarrow_+, \uparrow, \uparrow^+)$;

but those bounds may not be optimal.

6. CONCLUSIONS

In this work, we studied properties of Regular Tree Pattern Queries, a formalism proposed in [3] for testing equivalence of AXML systems.

We expanded RTPQ formalism and discovered its close connection with datalog. We established equivalence between the most robust variant: $\mathsf{RTPQ}(\downarrow, \downarrow_+, \uparrow, \uparrow^+)$ and linear monadic datalog. Using this result, we strengthened undecidability result of datalog containment in unions of conjunctive queries from [2]. In that paper, authors proved undecidability of containment over data trees labelled with symbols from finite alphabet of size at least 2. We showed that the problem is undecidable also for unlabelled data trees, as long as queries can use data constants, or equivalently, if the labelling alphabet is infinite.

We also investigated complexity of satisfiability, containment and satisfiability of boolean combinations for subclasses of $\mathsf{RTPQ}(\downarrow, \downarrow_+, \uparrow, \uparrow^+)$ with limited sets of axes. These results are summarized in Tables 1 and 2.

RWPQ	**SAT**	**$\subseteq$, BC-SAT**
$\rightarrow$	PSPACE	PSPACE
$\rightarrow, \leftarrow$	PSPACE	PSPACE
$\rightarrow, \rightarrow^+$	NEXPTIME	UNDEC.

Table 1: Results summary: words

RTPQ	**SAT**	**$\subseteq$, BC-SAT**
$\downarrow$	PTIME	PSPACE
$\downarrow, \uparrow$	PSPACE-hard	EXPTIME
$\downarrow, \downarrow_+$	PTIME	EXPSPACE
$\downarrow, \downarrow_+, \uparrow, \uparrow^+$	NEXPTIME-hard	UNDEC.

Table 2: Results summary: trees

Decidability of containment for $\mathsf{RTPQ}(\downarrow, \downarrow_+)$ solves an open problem from [3]: testing equivalence of simple AXML systems with arbitrary joins is decidable.

Future work. While we solved most of the problems for RTPQs there are still some remaining gaps. We established only hardness results for SAT of $\mathsf{RTPQ}(\downarrow, \downarrow_+, \uparrow, \uparrow^+)$ and $\mathsf{RTPQ}(\downarrow, \uparrow)$; upper bounds are missing.

More importantly, in the reductions we often relied on the arbitrary location of the out node. This way, many data values were passed between patterns. It would be interesting to see what happens if we restrict the queries by allowing the out node only in leafs/last positions of patterns. In this variant, only one data value can be passed between two patterns. We conjecture that under this restriction, containment is decidable.

7. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. ADDISON WESLEY Publishing Company Incorporated, 1995.

[2] Serge Abiteboul, Pierre Bourhis, Anca Muscholl, and Zhilin Wu. Recursive queries on trees and data trees. In *ICDT*, pages 93–104, 2013.

[3] Serge Abiteboul, Balder ten Cate, and Yannis Katsis. On the equivalence of distributed systems with queries and communication. In *ICDT*, pages 126–137, 2011.

[4] Michael Benedikt, Pierre Bourhis, and Pierre Senellart. Monadic datalog containment. In *ICALP (2)*, pages 79–91, 2012.

[5] Michael Benedikt, Wenfei Fan, and Floris Geerts. Xpath satisfiability in the presence of dtds. *J. ACM*, 55(2), 2008.

[6] Piero A. Bonatti. On the decidability of containment of recursive datalog queries - preliminary report. In *PODS*, pages 297–306, 2004.

[7] Diego Calvanese, Giuseppe De Giacomo, and Moshe Y. Vardi. Decidable containment of recursive queries. *Theor. Comput. Sci.*, 336(1):33–56, 2005.

[8] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic programming and databases*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[9] Surajit Chaudhuri and Moshe Y. Vardi. On the equivalence of recursive and nonrecursive datalog programs. In *PODS*, pages 55–66, 1992.

[10] Diego Figueira. Satisfiability of downward xpath with data equality tests. In *PODS*, pages 197–206, 2009.

[11] Haim Gaifman, Harry G. Mairson, Yehoshua Sagiv, and Moshe Y. Vardi. Undecidable optimization problems for database logic programs. *J. ACM*, 40(3):683–713, 1993.

[12] Georg Gottlob and Christoph Koch. Monadic datalog and the expressive power of languages for web information extraction. *J. ACM*, 51(1):74–113, 2004.

[13] Gerome Miklau and Dan Suciu. Containment and equivalence for a fragment of xpath. *J. ACM*, 51(1):2–45, 2004.

[14] Frank Neven and Thomas Schwentick. On the complexity of xpath containment in the presence of disjunction, dtds, and variables. *Logical Methods in Computer Science*, 2(3), 2006.

[15] Oded Shmueli. Equivalence of datalog queries is undecidable. *J. Log. Program.*, 15(3):231–241, 1993.