Containment of Data Graph Queries

Egor V. Kostylev
University of Edinburgh
ekostyle@inf.ed.ac.uk

Juan L. Reutter
PUC Chile
jreutter@ing.puc.cl

Domagoj Vrgoč
University of Edinburgh
domagoj.vrgoc@ed.ac.uk

ABSTRACT

The graph database model is currently one of the most popular paradigms for storing data, used in applications such as social networks, biological databases and the Semantic Web. Despite the popularity of this model, the development of graph database management systems is still in its infancy, and there are several fundamental issues regarding graph databases that are not fully understood. Indeed, while graph query languages that concentrate on topological properties are now well developed, not much is known about languages that can query both the topology of graphs and their underlying data.

Our goal is to conduct a detailed study of static analysis problems for such languages. In this paper we consider the containment problem for several recently proposed classes of queries that manipulate both topology and data: regular queries with memory, regular queries with data tests, and graph XPath. Our results show that the problem is in general undecidable for all of these classes. However, we find natural fragments that enjoy much better static analysis properties: the containment problem is decidable, and its computational complexity ranges from PSPACE-complete to EXPSPACE-complete. We also propose several extensions of the classes and study containment for them.

1. INTRODUCTION

Managing graph-structured data is one of the most active topics in the database community these days. Although first introduced in the eighties [15, 14], the model has recently gained popularity due to a high demand from services that find the relational model too restrictive, such as Social Networks, Semantic Web, crime detection networks, biological databases and many others. There are several vendors offering graph database systems [32, 17, 21] and a growing body of literature on the subject (for a survey see e.g. [2, 7, 40]).

In such applications the data is usually modelled as a graph, with each node describing one entity in the database, for example a user in a social network, and the edges of the graph representing various connections between nodes, such as friends in a social network, supervisor connection in a database modelling the structure of a company, etc. Nodes can have multiple types of connections, so usually each edge in the graph is labelled. Finally, nodes by themselves contain the actual data, modelled as traditional relational data with values coming from an infinite domain [2].

To query graph-structured data, one can, of course, use traditional relational languages and treat the model as a relational database. However, what makes graph databases attractive in modern applications is the ability to query intricate navigational patterns between objects, thus obtaining more information about the *topology* of the structure of the stored data, as well as the relations between the topology and data. Earliest graph query languages, such as regular path queries (RPQs) [15] and conjunctive regular path queries (CRPQs) [11, 14], concentrate on retrieving the topology of the graph and ignore the actual data stored. Such languages have been well studied in the previous decades and many extensions such as backward navigation [11], branching [6], or checks of nontrivial relations amongst paths [3] were defined for them, allowing users to specify more complex patterns connecting nodes in a graph.

But purely navigational languages such as RPQs or CRPQs cannot reason on the data stored in the nodes. Thus such data was usually queried using relational languages, without a way of specifying the interplay between the data stored and various navigational patterns connecting the data.

This interplay is indeed a requirement in many applications using graph-structured data. For example, in a database modelling the inner workings of a company one might be interested in finding chains of people of the same age connected by professional links, or in a social network one could check if there is a sequence of friends, all of which like the same type of music. Recently, several languages that can handle such queries have been proposed [29, 28, 27] and they were all built on the idea of extending RPQs, or some variation thereof, with the ability to reason about data values that appear along the navigated path.

Our goal is to study static analysis aspects of this new generation of graph query languages. We concentrate on the query containment problem, which is the problem of deciding, given two queries in some graph language, whether the answer set of the first query is contained in the answer set of the second one. Deciding query containment is a fundamental problem in database theory, and it is relevant to

several complex database tasks such as data integration [26], query optimisation [1], view definition and maintenance [22] and query answering using views [13]. The importance of this problem has motivated sustained research for relational query languages (see e.g. [1]), XML query languages (see e.g. [37]) and even extensions of RPQs and other graph query languages [18, 11, 5, 3]. Although we primarily concentrate on containment, the techniques used here can easily be adapted to deal with other similar problems, such as satisfiability or equivalence of queries.

While query containment of navigational graph languages is well understood by now [11, 3, 18], no detailed study has been conducted for query languages that deal both with navigational and data aspects of graph databases. In this work we concentrate on three such languages. Namely, we consider *regular queries with memory* (*RQMs*), *regular queries with data tests* (*RQDs*) both introduced in [29], as well as the recent adaptation [28] of the widely used XML query language XPath to the graph setting, which is called *graph* XPath (or, GXPath).

The intuition behind RQMs is that one can navigate through a graph in the same way as with RPQs, but along the path it is also possible to store a data value into a register and later on compare it with another value encountered further on the path. This idea is very similar to the one of register automata [25, 33] and in fact one can show that these two formalisms are equivalent [30]. RQDs operate in a similar fashion, but storing and comparing values adheres to a more strict stack-like discipline, so they enjoy much better evaluation properties. Lastly, the language of GXPath allows one to define patterns in the graph that are not necessarily just paths, as it is in the cases of RQMs and RQDs, and also accommodates the ability to test some data values in these patterns for equality and inequality.

Contributions By using equivalence of RQMs with register automata, we obtain our first result: the problem of deciding whether one RQM is contained in another RQM is undecidable. This, of course, opens up the question of fragments of the language that do have decidable containment problem. The class of *positive RQMs* is one of such fragments, in which we allow tests of data values only for equality, but not for inequality. We show, that the problem of positive RQMs query containment is decidable, and, in fact, EXPSPACE-complete.

Next we move onto the class of RQDs, which was shown to be strictly contained in the class of RQMs [29]. The imposed restrictions to RQMs are quite heavy, and computational complexity of query evaluation drops by almost one exponent when we consider RQDs instead of RQMs. For this reason one may expect the containment problem to be decidable for RQDs. On the contrary, as we show, it remains undecidable even in this restricted scenario. However, this changes once again when we consider *positive RQDs*, for which a PSPACE algorithm for testing containment is obtained. This is the best possible bound for any extension of RPQs, since their query containment is already PSPACE-hard [12].

A common assumption when considering graph languages is that edges can be traversed in both directions. Indeed, the authors in [10, 11] argue that any practical query mechanism for graphs should incorporate this functionality, as there are many scenarios when backward navigation is required. It is therefore natural to study what happens when RQMs and RQDs are extended with the inverse operator. This gives rise to two new classes of languages, called 2RQMs and 2RQDs, respectively. Remarkably, we show that adding this operator carries no extra computational cost with respect to query evaluation. However, it does make a big difference for containment, as even the subclass that allows only positive data comparisons has undecidable query containment problem.

Finally, we consider GXPath and its various dialects. This language has recently attracted some attention because it provides considerable expressive power while maintaining good query evaluation properties (in particular, the combined complexity is in polynomial time). However, with respect to containment the story is different: even the navigational fragment that uses no data value comparisons has undecidable containment problem. Although this bound follows from some folklore results on satisfiability of the three variable fragment of first order logic, we could not find a formal proof of this fact and, hence, provide a self-contained one by a reduction from the tiling problem.

The reason for the undecidability of GXPath is the presence of a powerful negation operator that allows complementation of binary relations. We show, that if one excludes such negation from the language, then containment becomes decidable (EXPSPACE-complete). Such a language is in fact close to a propositional dynamic logic (PDL), whose containment is also known to be EXPSPACE-complete [24].

The classes above do not consider any means for data values tests in GXPath queries. Following [28], we consider an extension of these classes with an operator to test whether data values at the beginning and at the end of a path are same or different. We arrive at a language that can simulate all RQDs [28]. Thus, our previous results imply that allowing for inequality tests immediately leads to undecidability of the containment problem.

Hence, the possible way to reach decidability is to consider GXPath queries that allow only equalities between data values. Whether or not the containment problem is decidable for this class, it promises to be a challenging task, worthy of an its own research line. Indeed, even for the case of trees with data, some similar problems are still open [8], and the ones that have been solved usually require very intricate techniques [31, 16].

Organization In Section 2 we formally define the data model and the problem studied. In Sections 3 and 4 we introduce RQMs and RQDs, respectively, and study their containment problems. In Section 5 we show how these classes

can be extended with inverses, and turn our attention to GX-Path in Section 6. We conclude with some remarks about future work in Section 7. Due to space limitations, most of the proofs are only sketched, and complete versions can be found in the appendix.

2. PRELIMINARIES

Data graphs Let Σ be a finite alphabet of *labels* and $\mathcal{D}$ be an infinite set of *data values*. A *data graph* over labels Σ and data values $\mathcal{D}$ is a triple $\langle V, E, \rho\rangle$, where:

- V is a finite set of nodes,
- $E \subseteq V \times \Sigma \times V$ is a set of labelled edges, and
- $\rho : V \to \mathcal{D}$ is a function that assigns a data value to each node in the graph.

An example of a data graph is shown in Figure 1. If data values are not important, we disregard ρ and only talk about *graphs* $\langle V, E\rangle$ over Σ.

Regarding data values, this paper follows [29, 28] and the standard convention for data trees (as a model for XML), and assumes that data values are attached to nodes. There are of course other possibilities, but they are all essentially equivalent. We also assume that each node is assigned with a single data value. This is not a real restriction, since tuples of attributes can be modelled by a set of edges, each of which is labeled with an attribute name and connects the current node to a new node with a data value for the corresponding attribute.

Paths A *path* π between nodes v_1 and v_n in a graph $\langle V, E\rangle$ is a sequence

$$v_1 a_1 v_2 a_2 v_3 \dots v_{n-1} a_{n-1} v_n,$$

such that each (v_i, a_i, v_{i+1}), for $1 \le i < n$, is an edge in E.

The *label* of the path π is the word $a_1 \dots a_{n-1}$ obtained by reading the edge labels appearing along this path.

Queries The default core of each query language for graphs are *regular path queries* (or *RPQs*) which are just regular languages over Σ, usually defined by regular expressions. The *evaluation* $[\![e]\!]^G$ of an RPQ e over a graph G, is the set of all pairs (v_1, v_2) of nodes in G for which there exists a path from v_1 to v_2 with the label from the language of e.

There are a number of extensions of RPQs proposed in the literature. In this paper we concentrate on those that are capable of dealing with data values. Also, all of the queries we study are binary queries, i.e. such that their answers are sets of pairs of nodes. We denote by $[\![e]\!]^G$ the answer of a query e over a data graph G.

Containment A query e_1 is *contained* in a query e_2 (written $e_1 \subseteq e_2$) if for every data graph G over Σ and $\mathcal{D}$ we have that

$$[\![e_1]\!]^G \subseteq [\![e_2]\!]^G.$$

The queries e_1 and e_2 are *equivalent* (written $e_1 \equiv e_2$) iff $[\![e_1]\!]^G = [\![e_2]\!]^G$ for every G.

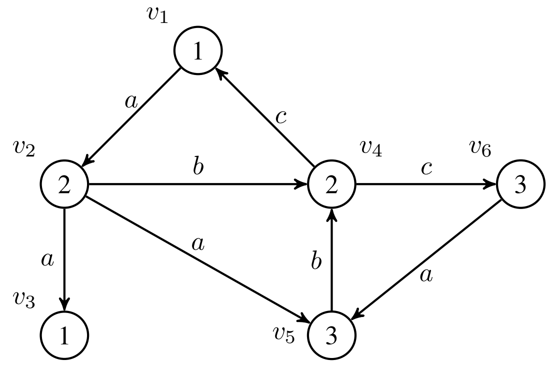


Figure 1: A data graph over labels $\{a, b, c\}$ and natural numbers as data values, in which nodes are v_i, $1 \le i \le 6$.

The containment and equivalence are at the core of many static analysis tasks, such as query optimisation. All the classes of queries considered in this paper are closed under union, so these two problems are easily interreducible: $e_1 \equiv e_2$ iff they are contained in each other, and $e_1 \subseteq e_2$ iff $e_1 \cup e_2 \equiv e_2$. That is why we concentrate just on the first and consider the following decision problem parametrized by a class of queries $\mathcal{Q}$.

CONTAINMENT ($\mathcal{Q}$)	
Input:	Queries e_1 and e_2 from $\mathcal{Q}$.
Question:	Is e_1 contained in e_2?

The semantics of RPQs is defined for graphs, but it is straightforward to see that for any two RPQs e_1 and e_2, we have that $e_1 \subseteq e_2$ if and only if the language accepted by the regular expression e_1 is contained in the language accepted by e_2 [12]. From this fact we obtain that containment of RPQs is PSPACE-complete, following the classic result that containment of regular expressions is PSPACE-complete. Since all of the classes of queries studied in this paper are extensions of RPQs, this establishes a lower bound for containment of any of these classes.

3. REGULAR QUERIES WITH MEMORY

Regular queries with memory, or *RQMs* for short, were first introduced in [29] as a formalism for querying data graphs that allows data comparisons while navigating through the structure of the graph (where they were called *regular expressions with memory*). They are based on *register automata*, an extension of finite-state automata for words over infinite alphabets (see, e.g. [33, 29] for a detailed description).

The idea of RQMs is the following. They can store data values in a number of named *registers*, while parsing the input graph according to a specified regular navigation pattern. Also, they can compare the current data value with values that had previously been stored. An example of an RQM is the expression $\downarrow x.a^+[x^=]$, which returns all pairs (v, v') of

nodes in a graph that have the same data value and are connected by a path labelled only with a's. Intuitively the expression works as follows: it first stores the data value of a node v into register x, and after navigating an a-labelled path, it checks that the node v' at the end of this path has the same data value as the first node. This check is done via test $x^=$ which makes sure that the data value of v' is the same as the one stored in register x.

The proposal of RQMs as a formalism for querying data graphs was motivated not only by their ability to handle data values, but also by the low computational complexity of their evaluation: it is PSPACE-complete in general, and NLOGSPACE-complete if the query is fixed (i.e., in *data complexity*) [29]. Hence, their query evaluation is essentially the same as for first-order or relational algebra queries.

3.1 Syntax and Semantics of RQMs

Let X be a countable set of *registers*. We will denote them by letters x, y, z, etc. A *condition* over X is a positive boolean combination of atoms of the form $x^=$ or $x^{\neq}$, for $x \in X$.

DEFINITION 3.1. *A* regular query with memory *(or* RQM*) over an alphabet of labels Σ and registers X is an expression satisfying the grammar*

$$e \;:=\; \varepsilon \mid a \mid e \cup e \mid e \cdot e \mid e^+ \mid e[c] \mid \downarrow x.e \qquad (1)$$

where ε is the empty word, a ranges over labels, x over registers, and c over conditions.

Before formally defining the semantics, let us give some examples of RQMs and explain their intuitive meaning.

EXAMPLE 3.2.

1. The RQM $\downarrow x.(a[x^=])^+$ returns all pairs of nodes connected by a path, along which all edges are labelled a and all data values are equal. The evaluation starts with $\downarrow x$, which stores the first data value into register x. The subexpression $(a[x^=])^+$ then checks that each subsequent label along the path is a, and that the data value of each node on this path is equal to the one of the first node (this is done by comparison with the value stored in register x). The fact that this subexpression is in the scope of $^+$ indicates that the length of the sequence of checks is of arbitrary length.
2. The RQM $\downarrow x.(a[x^{\neq}])^+$ returns all pairs of nodes connected by a path where all edges are labelled with a and the first data value is different from all other data values. It works analogously as the expression above, except that it checks for inequality.
3. The RQM $\downarrow x.(abc)^+[x^{\neq}]$ returns all pairs of nodes connected by a path, whose label is of the form $abc \ldots abc$, and the first data value is different from the last. Note that the order of $^+$ and condition is different from the previous examples: the condition is checked only once, after verifying that the label is in $(abc)^+$, i.e. at the end of the path. $\square$

$$\begin{array}{lcl}
\mathcal{H}^G(\varepsilon) & = & \{(s,s) \mid s \text{ is a state}\},\\
\mathcal{H}^G(a) & = & \{((v,\lambda),(v',\lambda)) \mid (v,a,v') \in E\},\\
\mathcal{H}^G(e_1 \cup e_2) & = & \mathcal{H}^G(e_1) \cup \mathcal{H}^G(e_2),\\
\mathcal{H}^G(e_1 \cdot e_2) & = & \mathcal{H}^G(e_1) \circ \mathcal{H}^G(e_2),\\
\mathcal{H}^G(e^+) & = & \mathcal{H}^G(e) \cup \mathcal{H}^G(e \cdot e) \cup \ldots,\\
\mathcal{H}^G(e[c]) & = & \{((v,\lambda),(v',\lambda')) \mid \\
 & & \quad ((v,\lambda),(v',\lambda')) \in \mathcal{H}^G(e) \text{ and } (\rho(v'),\lambda') \models c\},\\
\mathcal{H}^G(\downarrow x.e) & = & \{((v,\lambda),(v',\lambda')) \mid \\
 & & \quad ((v,\lambda),(v',\lambda')) \in \mathcal{H}^G(e) \text{ and } \lambda(x) = \rho(v)\}.
\end{array}$$

Table 1: Definition of the function $\mathcal{H}^G$ with respect to a data graph G.

To define what it means for a data value to satisfy a condition we need the following notion. An *assignment* of registers X is a partial function λ, from X to the set of data values $\mathcal{D}$. Intuitively, an assignment models the current state of the registers at some point of computation, with some registers containing stored data values, and some still being empty. Formally, a data value d and an assignment λ *satisfy* a condition $x^=$ (or $x^{\neq}$) iff $\lambda(x)$ is defined and $d = \lambda(x)$ (or $d \neq \lambda(x)$, correspondingly). This satisfaction relation is denoted $\models$ and extended to general conditions in the straightforward way.

Given a data graph G and a set of registers X, a *state* is a pair consisting of a node of G and an assignment of X.

The semantics of RQMs over a data graph $G = \langle V, E, \rho \rangle$ is defined in terms of function $\mathcal{H}^G$, which binds each RQM with a set of pairs of states. The intuition of the set $\mathcal{H}^G(e)$, for some RQM e, is as follows. Given states $s = (v, \lambda)$ and $s' = (v', \lambda')$, the pair (s, s') is in $\mathcal{H}^G(e)$ if there exists a path w from v to v', such that the expression e can parse w assuming that the registers are initialized according to λ, modified and compared as dictated by e, and finished according to λ'.

Formally, given a data graph $G = \langle V, E, \rho \rangle$, the function $\mathcal{H}^G$ is constructed by the inductive definition in Table 1.

The symbol $\circ$ in the table refers to the usual composition of binary relations:

$$\mathcal{H}^G(e_1) \circ \mathcal{H}^G(e_2) = \{(s_1, s_3) \mid \\ \exists s_2 \text{ s.t. } (s_1, s_2) \in \mathcal{H}^G(e_1) \text{ and } (s_2, s_3) \in \mathcal{H}^G(e_2)\}.$$

Finally, the *evaluation* $[\![e]\!]^G$ of an RQM e over a data graph G is the following set of pairs of nodes in G:

$$\{(v, v') \mid \exists \lambda' \text{ s.t. } ((v, \bot), (v', \lambda')) \in \mathcal{H}^G(e)\},$$

where $\bot$ is the empty assignment.

EXAMPLE 3.3. Consider the evaluations of expressions from Example 3.2 over the data graph from Figure 1:

1. the evaluation of $\downarrow x.(a[x^=])^+$ is $\{(v_6, v_5)\}$;
2. the evaluation of $\downarrow x.(a[x^{\neq}])^+$ is $\{(v_1, v_2), (v_1, v_5), (v_2, v_5), (v_2, v_3)\}$;

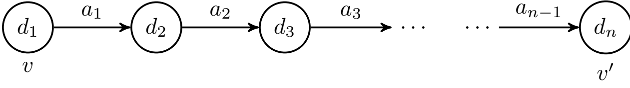


Figure 2: The data graph G_w corresponding to the data word $w = d_1a_1d_2 \ldots a_{n-1}d_n$ (some node identifiers are omitted).

3. the evaluation of $\downarrow x.(abc)^+[x^{\neq}]$ contains $(v_1, v_6), (v_6, v_1), (v_2, v_1)$ and (v_2, v_6) (but not (v_6, v_6)). $\square$

3.2 From Graphs to Words

As we mentioned in the preliminaries, standard algorithms for containment of RPQs rely on a simple fact that two RPQs are contained if and only if the regular languages they define are contained [12]. In this section we exhibit a similar behaviour for RQMs.

Data words are a widely studied extension of words over finite alphabets [36], in which every position carries not only a label from the finite alphabet Σ, but also a data value from the infinite $\mathcal{D}$. However, just for uniformity of presentation, we follow [29] and opt to the following essentially equivalent definition, by which data values are attached not to positions in a word, but "between" them.[1]

DEFINITION 3.4. *A* data word *over a finite alphabet of labels Σ and infinite set of data values $\mathcal{D}$ is a sequence $d_1a_1d_2a_2 \ldots a_{n-1}d_n$, where $n > 0$, $a_i \in \Sigma$, for each $1 \leq i < n$, and $d_i \in \mathcal{D}$, for each $1 \leq i \leq n$.*

Every data word $w = d_1a_1d_2 \ldots a_{n-1}d_n$ can be easily transformed to a data graph G_w, consisting of n different nodes with data values $d_1, \ldots, d_n$, respectively, consequently connected by edges labeled with $a_1, \ldots, a_{n-1}$, as illustrated in Fig. 2.

The semantics of RQMs over data words is defined in the straightforward way: a data word w is *accepted* by an RQM e iff $(v, v') \in [\![e]\!]^{G_w}$, where v and v' are the first and the last nodes of G_w. The set of all data words, accepted by an RQM e is denoted $\mathcal{L}(e)$.

Coming back to graphs, each path

$$v_1a_1v_2a_2v_3 \ldots v_{n-1}a_{n-1}v_n,$$

in a data graph $\langle V, E, \rho \rangle$ has the *corresponding* data word

$$\rho(v_1)a_1\rho(v_2)a_2\rho(v_3) \ldots \rho(v_{n-1})a_{n-1}\rho(v_n).$$

As noted in [29], for each RQM e, data graph G, and nodes v, v' of G, it holds that $(v, v') \in [\![e]\!]^G$ iff there exists a path between v and v' such that its corresponding data word is accepted by e. From this we have the following straightforward fact about containment of RQMs, similar to the property of RPQs, mentioned in the preliminaries.

[1] In [29] to distinguish this notion from the original, the term "data path" was used.

PROPOSITION 3.5. *Given two RQMs e_1 and e_2, it holds that $e_1 \subseteq e_2$ iff $\mathcal{L}(e_1) \subseteq \mathcal{L}(e_2)$.*

In the proposition above $e_1 \subseteq e_2$ is defined on data graphs, but $\mathcal{L}(e_1)$ and $\mathcal{L}(e_2)$ are sets of data words.

3.3 Containment of RQMs

We now turn to the containment problem for RQMs. Unfortunately, as the following theorem shows, the power that RQMs gain through its data manipulation mechanism, comes with a high price for static analysis tasks.

THEOREM 3.6. *The problem* CONTAINMENT *(*RQMs*) is undecidable.*

This fact immediately follows from undecidability of the containment problem for register automata ([33]), which are known to be equivalent to RQMs evaluated on data words ([29, 30]), and Proposition 3.5.

The theorem above naturally leads to question of finding decidable subclasses. It is known that testing containment of an expression using at most one register in an expression using at most two registers is decidable [33].

However, we concentrate on *positive RQMs*, i.e. those RQMs, that use only atoms of the form $x_i^=$ in the conditions. In [38] it was shown that the containment of positive RQMs is decidable, but no complexity bounds were given. The following theorem fills this gap.

THEOREM 3.7. *The problem* CONTAINMENT *(*positive RQMs*) is* EXPSPACE*-complete.*

PROOF SKETCH. Let e_1 and e_2 be two RQMs over Σ. For the EXPSPACE upper bound, assume that $e_1 \not\subseteq e_2$. Then by Proposition 3.5 there is a data word w and graph G_w such that $(v, v') \in [\![e_1]\!]^{G_w}$, v and v' being the first and last nodes of G_w. By definition of the semantics of RQDs, one can assign to each node in G_w a particular assignment for the registers in e_1, according to the states of relation $\mathcal{H}^{G_w}(e_1)$. We can then show that one can always have such a graph G_w satisfying, in addition, the following: two nodes u and u' of G_w have the same data value d if and only if the assignment for the registers in all nodes in the path from u to u' assign d. Combining this property with the fact that RQMs are equivalent to register automata [29], we show that, to check whether there is a data word that belongs to $\mathcal{L}(e_1)$ but does not belong to $\mathcal{L}(e_2)$, it suffices to guess a word w with the above properties, and simulate the automata equivalent to e_1 and e_2 without storing the precise information, but rather storing which registers in e_1 and e_2 store the same data values, and which one store different ones. In other words, we can build a transition system whose set of states store roughly the information of the current state of the automata equivalent to e_1 and e_2, plus the equality class formed by the data values stored in the registers of e_1 and e_2. The size of this system is double exponential in e_1 and e_2, but a reachability test can be performed by a standard on-the-fly procedure.

Hardness is by a reduction from the acceptance problem of a Turing Machine that works in EXPSPACE. The reduction is similar as the one in [4] (see Theorem 6), except that the gadgets in this proof are constructed by taking advantage of registers, instead of the variable assignments used there. □

The proof relies on the fact, that the set of registers X is unbounded. Moreover, for each class of *n-bounded positive RQMs*, i.e. positive RQMs which can use at most n registers, we have the following immediate corollary.

COROLLARY 3.8. *Let n be a natural number. The problem* CONTAINMENT *(n-*bounded positive RQMs*) is* PSPACE*-complete.*

Hence, positive RQMs are a natural subclass of RQMs with decidable query containment. However, when comparing the complexity with the one for RPQs, we see that allowing positive data test comparisons results in an exponential jump. In the following section we consider another class of queries extending RPQs, which also allows data value comparisons, but in a more restricted way than RQMs. As we will see, the positive subclass of this class has the same complexity of query containment as RPQs.

4. REGULAR QUERIES WITH DATA TESTS

Looking for classes of queries handling data values, but having better query answering properties than RQMs, the authors of [29] introduced *regular queries with data tests*, or *RQDs* for short (these were called *regular expressions with equality* in the original paper). An example of such a query is the expression $a(b^+)_=c$, whose intention is to return all pairs of nodes connected by a path, such that its label is $ab\dots bc$ and the data values before and after the sequence of b's are the same.

All RQDs are RQMs, but the usage of registers is restricted: each stored data value can be retrieved and compared only once, and the order of these storing and retrieving operations is not arbitrary, but on the "last in, first out" basis. The data complexity of RQDs' evaluation is the same as for RQMs – in NLOGSPACE, but the combined complexity is much better, and, if fact, even tractable, in PTIME [29].

4.1 Syntax and Semantics of RQDs

The syntax for RQDs can be defined in a direct, much simpler way than for RQMs, without even mentioning registers and conditions.

DEFINITION 4.1. *A* regular query with data tests *(or* RQD*) over an alphabet of labels Σ is an expression satisfying the grammar*

$$e \;:=\; \varepsilon \mid a \mid e \cup e \mid e \cdot e \mid e^+ \mid e_= \mid e_{\neq} \qquad (2)$$

where a ranges over labels.

Again, before the formal definition of semantics we give some examples of RQDs and their correspondence to RQMs.

$$\begin{array}{lcl}
[\![\varepsilon]\!]^G & = & \{(v,v) \mid v \in V\}, \\
[\![a]\!]^G & = & \{(v,v') \mid (v,a,v') \in E\}, \\
[\![e_1 \cdot e_2]\!]^G & = & [\![e_1]\!]^G \circ [\![e_2]\!]^G, \\
[\![e_1 \cup e_2]\!]^G & = & [\![e_1]\!]^G \cup [\![e_2]\!]^G, \\
[\![e^+]\!]^G & & \text{is the transitive closure of } [\![e]\!]^G, \\
[\![e_=]\!]^G & = & \{(v,v') \mid (v,v') \in [\![e]\!]^G, \rho(v) = \rho(v')\}, \\
[\![e_{\neq}]\!]^G & = & \{(v,v') \mid (v,v') \in [\![e]\!]^G, \rho(v) \neq \rho(v')\}.
\end{array}$$

Table 2: Semantics of RQDs with respect to a data graph G. The composition of binary relations is again denoted $\circ$.

EXAMPLE 4.2. Recall RQMs from Example 3.3 (here we consider the examples in different order for better understanding of the relation between RQMs and RQDs).

1. The RQM $\downarrow x.(abc)^+[x^{\neq}]$ can be written as the RQD $((abc)^+)_{\neq}$: the first data value is stored, then the sequence of abc's is read, and then the value is retrieved and compared for inequality with the current one. Note that the stored value is used just once.

2. The RQM $\downarrow x.(a[x^=])^+$ can be written as the RQD $(a_=)^+$: the first data value is stored; then a is read; then the stored data value is retrieved and compared with the current one for equality; if successful, this current value (equal to the original!) is stored again, another a is read, and so on. If the parsing continues, then the current data value is always equal to the original one, even if we use each stored value just once.

3. Contrary to the previous case, it can be shown that the RQM $\downarrow x.(a[x^{\neq}])^+$ cannot be expressed as an RQD: indeed, after the first comparison the original data value is lost, and storing the current data value (different from the original) cannot help with correct comparison on the next step.

4. The RQM $\downarrow x.a \downarrow y.b[y^=]c[x^=]$ can be written as the RQD $(ab_=c)_=$. However, the very similar RQM $\downarrow x.a \downarrow y.b[x^=]c[y^=]$ is not expressible as RQD, because the data values have now to be retrieved in an order different from the reverse, which is the only one possible for RQDs. □

Due to the restrictions in the syntax of RQDs, their semantics also can be defined in much simpler way, in comparison with RQMs. The *evaluation* $[\![e]\!]^G$ of an RQD e over a data graph $G = \langle V, E, \rho\rangle$ is the set of all pairs (v_1, v_2) of nodes in V defined recursively in Table 2.

As Example 4.2 suggests, and as it is formally shown in [29], the class of RQDs is strictly contained in the class of RQMs. Indeed, to transform an RQD to RQM we just need to recursively replace each subexpression of the form $e_\sim$, $\sim \in \{=, \neq\}$, with the subexpression $\downarrow x.e[x^\sim]$, where x is a previously unused register. However, there are RQMs which cannot be transformed to RQDs, which is also justified by the lower complexity of query evaluation.

Similarly to RQMs, each RQD also defines a language of data words. A data word w is *accepted* by an RQD e iff $(v, v') \in [\![e]\!]^{G_w}$, with G_w as in the Figure 2. The set of all data words accepted by an RQD e is denoted $\mathcal{L}(e)$. It is easy to see that for each RQD e, data graph G and nodes v, v' in G, it holds that $(v, v') \in [\![e]\!]^G$ iff there exists a path between v and v' such that its corresponding data word is accepted by e. This allows us to show an analogue of Proposition 3.5, thus reducing query containment to language containment.

PROPOSITION 4.3. *Given two RQDs e_1 and e_2, it holds that $e_1 \subseteq e_2$ iff $\mathcal{L}(e_1) \subseteq \mathcal{L}(e_2)$.*

4.2 Containment of RQDs

RQDs were originally introduced as a restriction of RQMs that enjoys much better query evaluation properties. In light of this result, one might also hope for good behaviour when query containment is considered. Surprisingly, the following theorem shows that this is not the case.

THEOREM 4.4. *The problem* CONTAINMENT *(*RQDs*) is undecidable.*

PROOF SKETCH. The proof follows the idea of coding the Post correspondence problem by data words from [33]. However, the expressions used there are RQMs and they rely on the fact that one can store a data value and then compare it with a value encountered later with no restrictions. This, on the other hand, is not possible when dealing with RQDs, since testing for (in)equality must adhere to the first-in-last-out discipline. The trick used to circumvent this is based on the observation that part of the coding from [33] can be reversed, thus allowing us to nest data value tests as dictated by the syntax of RQDs. □

This naturally opens the search for subclasses of RQDs with decidable containment problem. Similarly to positive RQMs, we now consider the class of *positive RQDs*, i.e. RQDs where subexpressions of the form $e_{\neq}$ are not allowed. We can obtain a positive RQM from a positive RQD by the described above procedure that transforms an RQD into an RQM. Hence, we again have a strict containment of the corresponding classes, and from Thm. 3.7 we conclude that containment of RQDs is decidable and in EXPSPACE. However, the following theorem says that we can perform even better, in fact, the best possible in light of the PSPACE lower bound for plain RPQs.

THEOREM 4.5. *The problem* CONTAINMENT *(*positive RQDs*) is* PSPACE*-complete.*

PROOF SKETCH. The hardness follows from the bounds for RPQs, so next we give an idea of an NPSPACE (and, hence, PSPACE) algorithm which decides whether $\mathcal{L}(e') \subseteq \mathcal{L}(e)$ holds for positive RQDs e' and e.

Let's start with a simple NPSPACE algorithm for containment of RPQs: (1) transform the RPQs to NFAs A' and A without ε-transitions; (2) put a pebble to each of the initial states; (3) repeat moving randomly the single pebble in A' along transitions, in parallel moving all the pebbles in A along the transitions labelled the same as the current transition in A': if we have several options, the pebble multiplies, if a pebble cannot move, it is removed, if several pebbles meet, just one is left; (4) stop and fail if the pebble in A' is in a final state, but none of the pebbles in A are; stop and succeed if the space is exhausted. Essentially, the set of pebbles in A is the state in a subset construction, done "on the fly".

A naive adaptation of this algorithm to deal with data values can be as follows.
(a) Before transforming to NFAs, *normalise* e and e' such that none of the equality checks $()_=$ can be opened together and none of them can be closed together on any run. This can be done, essentially, by applying the rules

$$((e_1)_=e_2)_= \rightsquigarrow (e_1)_=(e_2)_=, \quad (e_1(e_2)_=)_= \rightsquigarrow (e_1)_=(e_2)_=,$$

and some others. After this, RQDs can be transformed to NFAs whose transitions have extra labels from the set $R = \{\emptyset, \uparrow, \downarrow, \downarrow\uparrow\}$, where $\uparrow$ means that an equality is opening, and $\downarrow$ that an equality is closing.
(b) Attach a stack of *reactions* to all the pebbles in A, where each reaction is a symbol from R. Then, during a run of an algorithm, if the pebble in A' moves along a transition with $\uparrow$, then every moved pebble in A pushes into its stack the extra label of the transition, but only if it is either $\emptyset$ or $\uparrow$; otherwise pebble does not pass (note, that the usual label matching is still checked). In turn, if the pebble in A' moves along a transition with $\downarrow$, then only those pebble pass, which popped extra label *pairs* with the label of the current transition: $\downarrow$ pairs with $\uparrow$, and $\emptyset$ pairs with itself. The extra label $\downarrow\uparrow$ can be handled similarly.

By this, e.g. $(ab_=c)_=$ is contained in $(abc)_=$ because the only pebble in the second NFA when reading b has stack $(\uparrow, \emptyset)$ and the current label is pairing $\emptyset$. The same $(ab_=c)_=$ is contained in $ab_=c$, because, after b the stack is $(\emptyset, \uparrow)$ and the label is pairing $\downarrow$, but it is not contained in $(ab)_=c$, because they are $(\uparrow, \emptyset)$ and not pairing $\downarrow$.

Such an adaptation would work, but it has space issues.

First, the normalisation step can cause an exponential blow-up, if nested simultaneously opened or closed equalities are combined with $\cup$ operation. So, a PSPACE algorithm should not apply the rules above, but deal with such situations on the fly: e.g. we may allow a pebble in A to pass through opening an equality, but only with a condition that this equality will be closed together with the previous one.

Second, and more serious problem is that even if the depth of each stack is bounded by the depth of the equality tests nesting in A', the number of different stacks is exponential. In fact, there are examples where exponentially big set of pebbles with different stacks are on the same state in A at some point of a run. However, such a set is *never arbitrary*, and lots of information in the stacks can be shared: if a stack

can be seen as a unary tree, then every set of such trees which appears on a run can be represented as a *dag*, whose width is polynomial.

By carefully exploiting the ideas above we describe a desired PSPACE algorithm in the appendix. □

5. LANGUAGES WITH INVERSE

RQMs and RQDs are recent, but established extensions of RPQs which manage data values. However, as noted in [11], RPQs by themselves lack a very natural construction for navigation through the structure of graphs—namely, the *inverse* operator. Indeed, consider for example a genealogy graph over a single $parent$ label, such as the one presented in Figure 3.

We assume that nodes represent people and data values are their names. A natural query over this graph, which does not deal with data values, would be to ask for all pairs of siblings. This, however, is clearly not expressible as an RPQ. On the other hand, it can be written as $parent^{-}parent$, where '$^{-}$' is the inverse operator, which traverses edges *backwards*. This query will retrieve e.g. (v_2, v_4) from the graph in Figure 3, since these nodes have a common parent v_1.

The class of queries enriching RPQs with inverse, called *2RPQs*, was introduced in [11], where it was shown that even with this extension query evaluation remains the same as for RPQs (namely NLOGSPACE-complete). Moreover, in [12] the authors also show that query containment is as efficient as for plain RPQs (namely PSPACE-complete).

In this section we consider the extensions of RQMs and RQDs with the inverse operator, called *2RQMs* and *2RQDs* respectively. As far as we are aware, these languages have never been formally investigated, but we believe that they are natural and intuitive formalisms for querying data graphs. For example, one query of interest in our genealogy database might be to retrieve all pairs of (blood) relatives with the same name. This can be easily done by the means of 2RQD $((parent^{-})^{+}parent^{+})_{=}$, which checks that two people have a common ancestor and ensures that they also have the same name. For example the pair (v_3, v_4) is an answer to this query in our sample graph.

The main focus of this paper is query containment. But since we introduce the languages of 2RQMs and 2RQDs here, after the formal definitions we first explore the complexity of query evaluation, and only after it proceed to the containment problems.

5.1 Definition and Evaluation of 2RQMs and 2RQDs

The syntax and semantics of 2RQMs is just a union of the syntax and semantics of 2RPQs and RQMs. The similar holds for 2RQDs.

DEFINITION 5.1. *A* 2-way regular query with memory*, or 2RQM, over alphabet of labels Σ and registers X, is an expression satisfying the grammar (1) in Definition 3.1 extended with a^{-} alternative, where a ranges over Σ.*

A 2-way regular query with data tests*, or 2RQD, over labels Σ is an expression satisfying (2) in Definition 4.1 extended with a^{-}.*

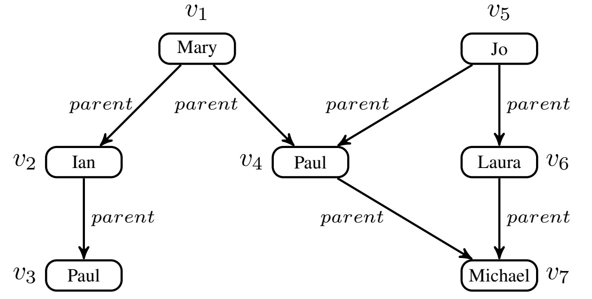


Figure 3: A genealogy database over the $parent$ label.

By this definition, 2RQDs restrict 2RQMs in the same way as RQDs restrict RQMs. The semantics of these languages extends their one-way analogs in the intuitive way. For 2RQMs, given a data graph $G = \langle V, E, \rho\rangle$, the function $\mathcal{H}^G$ extends the definition from Table 1 to the inverse construction as follows:

$$\mathcal{H}^G(a^{-}) \quad = \quad \{((v', \lambda), (v, \lambda)) \mid (v, a, v') \in E\}.$$

Then, the *evaluation* $[\![e]\!]^G$ of an 2RQM e over a data graph G stays the same as for RQMs.

Similarly, the *evaluation* $[\![e]\!]^G$ of a 2RQD e over a data graph $G = \langle V, E, \rho\rangle$ is obtained by adding the following rule to Table 2:

$$[\![a^{-}]\!]^G \quad = \quad \{(v', v) \mid (v, a, v') \in E\}.$$

As noted above, the complexity of 2RPQ evaluation is the same as for plain RPQs. Next we show that the same also holds for RQMs and RQDs with their two-way variants.

PROPOSITION 5.2. *The problem of deciding whether a pair of nodes belongs to $[\![e]\!]^G$ for a 2RQM e and data graph G is* PSPACE*-complete. The same problem is in* PTIME *if e is a 2RQD. If we assume that e is fixed the problem becomes* NLOGSPACE*-comlete.*

The proof of this proposition follows from the evaluation algorithms for RQMs and RQDs described in [29], and the observation that such two-way query can be viewed as an ordinary one-way query over the extended alphabet $\Sigma' = \Sigma \cup \{a^{-} \mid a \in \Sigma\}$. Then a pair (v, v') is an answer of e as a two-way query over a graph if and only if it is an answer of e as an one-way query, but over extended graph, which has (v', a^{-}, v) edge, for each edge (v, a, v').

5.2 Containment of 2RQMs and 2RQDs

The classic result by Calvanese et al. [12] states that one can add the inverse operator to RPQs and maintain not only

the same complexity of query evaluation, but also the same complexity of query containment. The proposition above gives a hope that the inverse functionality will not affect the complexity of containment of 2RQMs and 2RQDs as well. Of course, by the results of the previous sections, containment is undecidable when full languages are considered. Unfortunately, as we show next, decidability for positive RQMs does not propagate to their two-way variant.

The class of *positive* 2RQMs is defined as a subclass of 2RQMs that use only conditions built from atoms of the form $x^=$, but not $x^{\neq}$. Note that for 2RQMs we can no longer use language containment to check for query containment [12]. Indeed, it might be tempting to do the same as we did for Proposition 5.2, and reduce containment checking of two-way queries to containment of the same queries, but viewed as one-way queries over the extended alphabet. However, the second containment does not imply the first, because labels of the form a^- no longer denote only backward edges, but act as arbitrary labels. This leads to the following announced result.

THEOREM 5.3. *The problem* CONTAINMENT *(*positive 2RQMs*) is undecidable.*

PROOF SKETCH. The proof is by a reduction from the emptiness problem of stateless multihead automata, shown to be undecidable in [41]. Two way register automata are known to be able to simulate multihead automata [33], and the same can be shown for 2RQMs. However, such simulation requires both equalities and inequalities, so the proof does not follow directly from this work.

We do not simulate a stateless multihead automaton $\mathcal{A}$ directly, but rather simulate only the accepting runs. We define positive 2RQMs e_1 and e_2 such that $\mathcal{A}$ accepts no words if and only if $e_1 \subseteq e_2$. In our coding, a witness G_w for $e_1 \not\subseteq e_2$ represents a word belonging to $\mathcal{A}$. □

This surprising negative result can be a serious obstacle for using even positive 2RQMs in applications. A natural question here is whether containment becomes decidable when less expressive class of positive 2RQDs is considered. We leave a search for the answer for future work.

6. GRAPH XPATH

As we saw in the previous section, 2RQMs and 2RQDs extend RPQs not only with the constructs for data values comparisons, but also with an additional navigational feature. The language of *Graph XPath*, or GXPath for short, which was introduced in [28] as an adaptation of the widely used XML query language XPath to the graph setting, goes further in this direction, extending the classes considered above with even more elaborate navigational tools. For example, the GXPath query $a[\langle b^+\rangle]c$ retrieves all pairs (v, v') of nodes connected by a path labelled ac, such that the intermediate node on this path has an outgoing sequence of b-labelled edges. The end point of that sequence can be arbitrary, we are interested just in its existence. The pattern described by this query is illustrated in Fig 4.

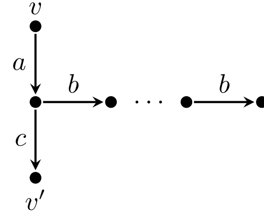


Figure 4: A pattern for GXPath query $a[\langle b^+\rangle]c$.

One consequence of this gain in navigational expressiveness is that we cannot always go from graphs to words as before: for instance, there are GXPath queries which are satisfiable on graphs, but not on words (like the one above). It means that we cannot hope for anything like Propositions 3.5 and 4.3, because query containment no longer corresponds to containment of languages.

Contrary to 2RQMs and 2RQDs, static analysis aspects of GXPath were not studied before even for purely navigational fragment $\mathsf{GXPath}_{\text{reg}}$ without data comparisons. That is why we start by exploring the containment problem for this fragment, and only after it proceed to various extensions with data tests.

Before proceeding to the formal details, it is worth to note, that the aforementioned class $\mathsf{GXPath}_{\text{reg}}$ essentially corresponds to the well studied formalism of *propositional dynamic logic*, or *PDL* [24], with negation on paths.

6.1 Syntax and Semantics of $\mathsf{GXPath}_{\text{reg}}$

As in XPath, formulas of $\mathsf{GXPath}_{\text{reg}}$ are divided into *path formulas*, returning pairs of nodes, and *node formulas*, returning single nodes. Since we are interested in extensions of RPQs (which are binary), we concentrate on path formulas, and node ones will play just auxiliary role. The formulas are defined by mutual recursion as follows.

DEFINITION 6.1. *Node formulas of* φ, ψ *of* $\mathsf{GXPath}_{\text{reg}}$ *and* path formulas α, β *are expressions satisfying the grammar*

$$\begin{array}{rcl}\varphi, \psi & := & \top \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \langle\alpha\rangle, \\ \alpha, \beta & := & \varepsilon \mid a \mid a^- \mid [\varphi] \mid \alpha \cup \beta \mid \alpha \cdot \beta \mid \overline{\alpha} \mid \alpha^+.\end{array} \quad (3)$$

Just by glancing the definition one immediately notices that $\mathsf{GXPath}_{\text{reg}}$ is a formalism much richer in navigational properties than RPQs: it allows inverse traversal of edges (the a^- operator), non-existence of paths (the $\overline{\alpha}$ operator), and testing for existence of (boolean combinations of) paths starting from the current node (the $[\varphi]$ operator). The formal semantics with respect to a graph $G = \langle V, E\rangle$ is given in Table 3: a node formula φ defines the set $[\![\varphi]\!]^G$ of nodes, and a path formula α defines the set $[\![\alpha]\!]^G$ of pairs of nodes.

Note that negation over path formulas is usually not included in the syntax of XPath when working on trees, since

$$
\begin{array}{rcl}
[\![\top]\!]^G &=& \{v \mid v \in V\}, \\
[\![\neg\varphi]\!]^G &=& V - [\![\varphi]\!]^G, \\
[\![\varphi \wedge \psi]\!]^G &=& [\![\varphi]\!]^G \cap [\![\psi]\!]^G, \\
[\![\varphi \vee \psi]\!]^G &=& [\![\varphi]\!]^G \cup [\![\psi]\!]^G, \\
[\![\langle\alpha\rangle]\!]^G &=& \{v \mid \exists v'\ (v,v') \in [\![\alpha]\!]^G\};
\end{array}
$$

$$
\begin{array}{rcl}
[\![\varepsilon]\!]^G &=& \{(v,v) \mid v \in V\}, \\
[\![a]\!]^G &=& \{(v,v') \mid (v,a,v') \in E\}, \\
[\![a^-]\!]^G &=& \{(v',v) \mid (v,a,v') \in E\}, \\
[\![[\varphi]]\!]^G &=& \{(v,v) \in G \mid v \in [\![\varphi]\!]^G\}, \\
[\![\alpha \cup \beta]\!]^G &=& [\![\alpha]\!]^G \cup [\![\beta]\!]^G, \\
[\![\alpha \cdot \beta]\!]^G &=& [\![\alpha]\!]^G \circ [\![\beta]\!]^G, \\
[\![\overline{\alpha}]\!]^G &=& V \times V - [\![\alpha]\!]^G, \\
[\![\alpha^+]\!]^G & & \text{is the transitive closure of } [\![\alpha]\!]^G.
\end{array}
$$

Table 3: The semantics of $\mathsf{GXPath}_{\mathrm{reg}}$. The symbol '$-$' stands for set-theoretic difference.

one can show that this class is closed under negation. This, however, is not the case for GXPath as shown in [28], so complementation is added to preserve close connection between XPath and first-order logic.

6.2 Containment of $\mathsf{GXPath}_{\mathrm{reg}}$ Queries

Analysing the expressive power of $\mathsf{GXPath}_{\mathrm{reg}}$ reveals that this class of queries is equivalent to the extension of first order logic with three variables (FO^3) with the transitive closure operator [28]. It is well known that satisfiability of FO^3 formulas is undecidable over arbitrary (possibly infinite) graphs, and it is folklore to assume that this bound is maintained for finite graphs, which we study in this paper. Since containment is a more general problem, than satisfiability, we have the following theorem.

THEOREM 6.2. *The* CONTAINMENT *(*$\mathsf{GXPath}_{\mathrm{reg}}$*) problem is undecidable.*

PROOF SKETCH. Since we could not find a formal proof of the aforementioned result about finite satisfiability of FO^3, we include a self contained proof in the appendix, as for all other theorems of this paper. The proof shows that even satisfiability problem for $\mathsf{GXPath}_{\mathrm{reg}}$ formulas is undecidable. To obtain this result we give a reduction from a variation of tiling problem from [23]. In particular we use the fact that the set $S_{notiling}$, of all finite sets of tile that can not tile the positive plane, and the set S_{period}, of all finite sets of tiles that can tile the plane periodically, are recursively inseparable.

Following the ideas from [19], we then show how to construct, for each finite set of tiles $\mathcal{T}$, a $\mathsf{GXPath}_{\mathrm{reg}}$ node formula $\gamma_{\mathcal{T}}$ such that satisfiability of $\gamma_{\mathcal{T}}$ implies that $\mathcal{T}$ can tile the positive plane, while the fact that $\mathcal{T}$ can tile the plane periodically implies that $\gamma_{\mathcal{T}}$ is satisfiable. Note that this shows that the set $S = \{\varphi : \exists G s.t. [\![\varphi]\!]^G \neq \emptyset\}$ contains the set $\{\gamma_{\mathcal{T}} : \mathcal{T} \in \mathcal{S}_{\text{period}}\}$ and is disjoint from $\{\gamma_{\mathcal{T}} : \mathcal{T} \in \mathcal{S}_{\text{notiling}}\}$. The fact that $S_{notiling}$ and S_{period} are recursively inseparable then implies that S can not be recursive, so satisfiability, and thus containment, of $\mathsf{GXPath}_{\mathrm{reg}}$ queries is undecidable.

To define the formula $\gamma_{\mathcal{T}}$ we heavily rely on the fact that $\mathsf{GXPath}_{\mathrm{reg}}$ can force loops in a graph, thus allowing us to check that that tiles are placed correctly and that the tiling can proceed from any point in the positive plane. $\square$

By analysing the proof one can also observe that the usage of the transitive closure operator $^+$ is restricted to edge labels only. Thus, we actually show that the satisfiability problem is already undecidable for the fragment of $\mathsf{GXPath}_{\mathrm{reg}}$, called $\mathsf{GXPath}_{\mathrm{core}}$ by analogy with the core fragment of XPath, which allows only a^+ and $(a^-)^+$ instead of α^+ in the grammar for path queries in (3). Note, that $\mathsf{GXPath}_{\mathrm{core}}$ does not contain RPQs any more, and in fact these two classes are incomparable [28].

Due to the before mentioned connection to PDL, we have a result on satisfiability of PDL with negation over finite models.

COROLLARY 6.3. *The satisfiability problem for PDL with negation on paths is undecidable over finite models, even in the absence of propositional variables.*

In fact, by carefully examining the proof, one can check that the use of negation is quite limited and that we only use intersection and the fact that $\mathsf{GXPath}_{\mathrm{reg}}$ can define the set of all pairs of mutually different nodes via the expression $\overline{\varepsilon}$. We are hoping that further adaptations of the proof could lead to solving the well know open problem of finite satisfiablity for PDL formulas with intersection [20].

As in the previous sections, we have the following question: what are the restrictions on $\mathsf{GXPath}_{\mathrm{reg}}$ that make containment decidable? The most natural candidates are of course the ones that forbid negation. Since we have two forms of negation, one on node formulas and another on path formulas, we consider two positive subclasses of $\mathsf{GXPath}_{\mathrm{reg}}$.

DEFINITION 6.4. *The* positive $\mathsf{GXPath}_{\mathrm{reg}}$, *denoted* $\mathsf{GXPath}_{\mathrm{reg}}^{\mathrm{pos}}$, *does not allow node formulas of the form* $\neg\varphi$ *and path formulas of the form* $\overline{\alpha}$ *in the grammar (3) of* $\mathsf{GXPath}_{\mathrm{reg}}$.

The path-positive $\mathsf{GXPath}_{\mathrm{reg}}$, *denoted* $\mathsf{GXPath}_{\mathrm{reg}}^{\text{path-pos}}$, *does not allow* $\overline{\alpha}$*, but keeps* $\neg\varphi$ *in the grammar.*

Note that, as opposed to the classes from previous sections, the word "positive" refers here to restrictions of navigational queries, but not for data manipulation.

A PSPACE upper bound for complexity of containment problem for $\mathsf{GXPath}_{\mathrm{reg}}^{\mathrm{pos}}$ queries is shown in [35]. Hence, this complexity is the same as for RPQs. Exploiting connections with PDL, we obtain the following result for the second, bigger class defined above.

THEOREM 6.5. *The decision problem* CONTAINMENT *(*$\mathsf{GXPath}_{\mathrm{reg}}^{\text{path-pos}}$*) is* EXPTIME*-complete.*

Data comparisions	RQD	RQM	2RQD	2RQM	$\mathsf{GXPath}_{\text{reg}}^{\text{pos}}$	$\mathsf{GXPath}_{\text{reg}}^{\text{path-pos}}$	$\mathsf{GXPath}_{\text{reg}}$
none	PSPACE-c*		PSPACE-c*		PSPACE-c*	EXPTIME-c	und.
positive	PSPACE-c	EXPSPACE-c	?	und.	?	?	und.
full	und.	und.	und.	und.	und.	und.	und.

Table 4: Summary of results. Results, known before, are marked with an asterisk. Some classes have synonyms, not given for clarity: i.e. RQDs and RQMs with no data comparisons are RPQs.

Note that this result gives us an upper bound of containment for *path-positive* $\mathsf{GXPath}_{\text{core}}$, i.e. the intersection of $\mathsf{GXPath}_{\text{core}}$ and $\mathsf{GXPath}_{\text{reg}}^{\text{path-pos}}$. We leave the precise bounds for *core* fragments for future work, as our focus in this paper is on queries extending RPQs.

6.3 Adding Data Values

Since $\mathsf{GXPath}_{\text{reg}}$ formulas are two-sorted, generally, there are two approaches to add data value comparisons. As we concentrate on path queries in this paper, next we consider the one which is in line with RQDs, studied in Sec. 4. The syntax of this new class $\mathsf{GXPath}_{\text{reg}}(\sim)$ extends the grammar (3) of $\mathsf{GXPath}_{\text{reg}}$ with path formulas of the form $\alpha_=$ and $\alpha_{\neq}$. The semantics over a data graph $G = \langle V, E, \rho\rangle$ enriches Table 3 in a way similar to semantics of RQDs:

$$\begin{aligned} [\![\alpha_=]\!]^G &= \{(v,v') \in [\![\alpha]\!]^G \mid \rho(v) = \rho(v')\},\\ [\![\alpha_{\neq}]\!]^G &= \{(v,v') \in [\![\alpha]\!]^G \mid \rho(v) \neq \rho(v')\}.\end{aligned}$$

Similarly to previous sections, we also consider subclasses $\mathsf{GXPath}_{\text{reg}}^{\text{pos}}(\sim)$ and $\mathsf{GXPath}_{\text{reg}}^{\text{path-pos}}(\sim)$ of $\mathsf{GXPath}_{\text{reg}}(\sim)$, the first of which does not allow node negations $\neg\varphi$ and path negations $\overline{\alpha}$, and the second one does not allow just path negations.

Another way to add data values tests would be to follow usual XPath and add node formulas $\langle \alpha = \beta\rangle$ to the syntax. The semantics of such a formula is all the nodes in the graph from which one can reach two nodes v' and v'' by following paths satisfying α and β respectively, such that $\rho(v') = \rho(v'')$. In [28] it was shown that such an extension of $\mathsf{GXPath}_{\text{reg}}$ is strictly contained in the defined above $\mathsf{GXPath}_{\text{reg}}(\sim)$.

Next we come to query containment for $\mathsf{GXPath}_{\text{reg}}(\sim)$ and its fragments. However, it is shown in [28], that even $\mathsf{GXPath}_{\text{reg}}^{\text{pos}}(\sim)$, i.e. the smallest subclass defined above, contains the class of RQDs. That is why we have the following corollary of Theorem 4.4.

COROLLARY 6.6. *The problems*

- CONTAINMENT *(*$\mathsf{GXPath}_{reg}^{pos}(\sim)$*),*
- CONTAINMENT *(*$\mathsf{GXPath}_{reg}^{path\text{-}pos}(\sim)$*) and*
- CONTAINMENT *(*$\mathsf{GXPath}_{reg}(\sim)$*)*

are undecidable.

The next step in the search for decidable fragments of GXPath would be to restrict data tests to equality tests of the form $\alpha_=$ only (i.e. forbid the form $\alpha_{\neq}$). We did such a restriction for RQDs and RQMs before.

From Theorem 6.2 we already know that containment for $\mathsf{GXPath}_{\text{reg}}(\sim)$ with such restriction is undecidable. However, results for similar fragments of RQDs give some hope that containment for $\mathsf{GXPath}_{\text{reg}}^{\text{path-pos}}(\sim)$ and $\mathsf{GXPath}_{\text{reg}}^{\text{pos}}(\sim)$ with such restrictions might be decidable. In future work we would like to extend our research in this direction, as well as study what happens in core fragments, where one might even be allowed to use inequality tests and still retain decidability of basic static analysis tasks.

7. CONCLUSION AND FUTURE WORK

After conducting a detailed study of main classes of queries for graphs with data we can conclude that the picture here is quite different than when static analysis of traditional navigational languages is considered. In particular, there is a sharp contrast between RPQs and CRPQs, which became a staple for navigational queries on graphs, where containment is decidable, and any of the proposed extension of RPQs that handle data values. Although undecidability for a class of RQMs comes as a no surprise, due to high complexity of query evaluation and powerful data manipulation mechanism, we have seen that even languages with good query evaluation properties can lead to undecidable query containment.

In particular the presence of inequality tests seems to be one of the major detractors here, although the ability to define complex navigational patterns can lead to undecidability as well. Thus to obtain decidable fragments, it is necessary to limit attention to purely positive variants of the language. The situation further complicates in the presence of inverse operator. We summarise all of the results in Table 4.

All of this shows that, although most of graph query languages are well established, there is still some fine tuning needed to define languages with desirable static analysis properties. In particular, we would like to fully understand the containment problem for all fragments of GXPath. Some results in this section give us hope that decidability could be obtained for positive fragments using only equality tests and for core fragments we did not consider here. Another approach is to weaken data tests and allow only standard XPath-like tests of the form $\langle \alpha = \beta\rangle$, which were shown to be weaker than the equality tests used here [28].

8. REFERENCES

[1] S. Abiteboul, R. Hull, V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
[2] R. Angles, C. Gutierrez. Survey of graph database models. *ACM Computing Surveys*, 40(1), 2008.
[3] P. Barceló, L. Libkin, A.W. Lin, P. Wood. Expressive languages for path queries over graph-structured data. *ACM TODS* 38(4) (2012).
[4] P. Barceló, J. Reutter, L. Libkin. Parameterized regular expressions and their languages.. *TCS* 474: 21–45 (2013).
[5] P. Barceló, L. Libkin, J. Reutter. Querying graph patterns. In *PODS'11*, pages 199–210.
[6] P. Barceló, J. Pérez, J. L. Reutter. Relative expressiveness of nested regular expressions. In *AMW'12*, pages 180–195.
[7] P. Barceló. Querying Graph Databases. In *PODS'13*.
[8] M. Benedikt, W. Fan, F. Geerts. XPath satisfiability in the presence of DTDs. In *J. ACM*, 55(2) (2008).
[9] Egon Börger, Erich Grädel, Y. Gurevich *The Classical Decision Problem.* Perspectives in Mathematical Logic, Springer, 2001.
[10] P. Buneman, S. B. Davidson, G. G. Hillebrand, D. Suciu A Query Language and Optimization Techniques for Unstructured Data. In *SIGMOD Conference 1996*, pages 505–516
[11] D. Calvanese, G. De Giacomo, M. Lenzerini, M.Y. Vardi. Containment of conjunctive regular path queries with inverse. In *KR'2000*, pages 176–185.
[12] D. Calvanese, G. De Giacomo, M. Lenzerini, M.Y. Vardi. Reasoning on regular path queries. *ACM SIGMOD Record*, 32(4):83–92, 2003.
[13] D. Calvanese, G. De Giacomo, M. Lenzerini, M.Y. Vardi. View-Based query answering and query containment over semistructured data. In *DBPL* 2001, pages 176–185.
[14] M. Consens, A. Mendelzon. GraphLog: a visual formalism for real life recursion. In *PODS'90*, pages 404–416.
[15] I. Cruz, A.O. Mendelzon, P. Wood. A graphical query language supporting recursion. In *SIGMOD'87*, pages 323–330.
[16] C. David, A. Gheerbrant, L. Libkin, W. Martens. Containment of pattern-based queries over data trees. *ICDT 2013*, pages 201–212.
[17] DEX query language. `http://www.sparsity-technologies.com/dex.php`.
[18] D. Florescu, A. Y. Levy, D. Suciu. Query Containment for Conjunctive Queries with Regular Expressions. *PODS'98*, pages 139–148.
[19] R. Goldblatt, M. Jackson. Well structured program equivalence is highly undecidable. *ACM Trans. Comput. Log.*, 13(3):26, 2012.
[20] S. Göller, M. Lohrey, C. Lutz. PDL with intersection and converse: satisfiability and infinite-state model checking. In *J. Symb. Log.*, 74(1): 279-314 (2009).
[21] The Gremlin graph traversal language. `http://gremlin.tinkerpop.com`.
[22] A. Gupta, I.S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. IEEE Data Eng. Bull., 18(2): 3–18, 1995.
[23] Y. Gurevich, I. Koryakov. Remarks on Berger's paper on the domino problem. In *Siberian Math. Journal*, 1972.
[24] D. Harel, D. Kozen, J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
[25] M. Kaminski, N. Francez. Finite memory automata. *TCS*, 134(2):329–363, 1994.
[26] M. Lenzerini. Data integration: a theoretical perspective. In *PODS*, 2002.
[27] L. Libkin, J. L. Reutter, D. Vrgoč. TriAL for RDF: Adapting Graph Query Languages for RDF Data. In *PODS*, 2013.
[28] L. Libkin, W. Martens, D. Vrgoč. Querying graph databases with XPath. In *ICDT*, 2013.
[29] L. Libkin, D. Vrgoč. Regular path queries on graphs with data. In *ICDT'12*, pages 74–85.
[30] L. Libkin, D. Vrgoč. Regular expressions for data words. *LPAR'12*, pages 274–288.
[31] G. Miklau, D. Suciu. Containment and equivalence for a fragment of XPath. *J. ACM*, 51(1): 2-45 (2004).
[32] The Neo4j Manual. `http://docs.neo4j.org`.
[33] F. Neven, T. Schwentick, V. Vianu. Finite state machines for strings over infinite alphabets. *ACM TOCL* 5(3): 403–435 (2004).
[34] J. Pérez, M. Arenas, C. Gutierrez. nSPARQL: A navigational language for RDF. *J. Web Sem.*, 8(4):255–270, 2010.
[35] J. L. Reutter. Containment of Nested Regular Expressions. CoRR abs/1304.2637 , (2013).
[36] L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In *CSL'06*, pages 41-57.
[37] T. Schwentick. XPath query containment. *ACM SIGMOD Record*, 33(1):101–109, 2004.
[38] A. Tal. *Decidability of Inclusion for Unification Based Automata*. M.Sc. thesis (in Hebrew), Technion, 1999.
[39] B. ten Cate, C. Lutz. The complexity of query containment in expressive fragments of XPath 2.0. In *J. ACM*, 56(6), 2009.
[40] P. Wood. Query languages for graph databases. *Sigmod Record*, 41(1):50–60, 2012.
[41] L. Yang, Z. Dang, O. H. Ibarra. On stateless automata and P systems. In *International Journal of Foundations of Computer Science*, 19(05), 1259–1276, 2008.

APPENDIX

In this appendix we give full proofs for all the propositions and theorems of the paper.

Proofs for Section 3

PROPOSITION 3.5. *Given two RQMs e_1 and e_2, it holds that $e_1 \subseteq e_2$ iff $\mathcal{L}(e_1) \subseteq \mathcal{L}(e_2)$.*

PROOF. In this proof we will use the following result from [28]: a pair of nodes (u, v) of a data graph G belongs to $[\![e]\!]^G$ if and only if there is a path from u to v such that its corresponding data word belongs to $L(e)$.

Assume first that $e_1 \subseteq e_2$. By definition it means that $[\![e_1]\!]^G \subseteq [\![e_2]\!]^G$, for every data graph G. Consider any data word $w = d_1 a_1 d_2 a_2 \ldots a_{k-1} d_k$ such that $w \in \mathcal{L}(e_1)$. By definition this means that $(v, v') \in [\![e_1]\!]^{G_w}$, where G_w is the data graph corresponding to w, as denoted in Figure 2. Then by our assumption we have $(v, v') \in [\![e_2]\!]^{G_w}$. From this and definition of $\mathcal{L}(e_2)$, it follows that $w \in \mathcal{L}(e_2)$, as desired.

On the other hand, suppose that $\mathcal{L}(e_1) \subseteq \mathcal{L}(e_2)$ and take any data graph G and two nodes $(v, v') \in [\![e_1]\!]^G$. By aforementioned fact there is a path from v to v' in G whose corresponding data word w belongs to $\mathcal{L}(e_1)$. Then by our assumption we have that $w \in \mathcal{L}(e_2)$, so using the same fact we get that $(v, v') \in [\![e_2]\!]^G$. □

THEOREM 3.6. *The problem* CONTAINMENT *(*RQMs*) is undecidable.*

PROOF. Since the class of RQDs, considered in Section 4 is a subclass of RQMs, this theorem is an immediate corollary of Theorem 4.4, which proof is given below. □

THEOREM 3.7. *The problem* CONTAINMENT *(*positive RQMs*) is* EXPSPACE*-complete.*

PROOF. We start with the upper bound. We need some auxiliary definitions and lemmas.

Register Data Word Automata It is more convenient to show the upper bound for register automata over data words, that we now define. Note that since we are using data words as in [29], we draw from their definition of register automata.

DEFINITION 8.1 (REGISTER DATA WORD AUTOMATA). *Let Σ be a finite alphabet, and k a natural number. A k-*register data path automaton *is a tuple $\mathcal{A} = (Q, q_0, F, \lambda_0, \delta)$, where:*

- *$Q = Q_w \cup Q_d$, where Q_w and Q_d are two finite disjoint sets of word states and data states;*
- *$q_0 \in Q_d$ is the initial state;*
- *$F \subseteq Q_w$ is the set of final states;*
- *$\lambda_0 \in \mathcal{D}^k$ is the initial configuration of the registers;*
- *$\delta = (\delta_w, \delta_d)$ is a pair of transition relations:*
  - *$\delta_w \subseteq Q_w \times \Sigma \times Q_d$ is the word transition relation;*
  - *$\delta_d \subseteq Q_d \times \mathcal{C}_k \times 2^{[k]} \times Q_w$ is the data transition relation.*

The intuition behind this definition is that since we alternate between data values and word symbols in data words, we also alternate between data states (which expect data value as the next symbol) and word states (which expect alphabet letters as the next symbol). We start with a data value, so q_0 is a data state, end with a data value, so final states, seen after reading that value, are word states.

In a word state the automaton behaves like the usual NFA (but moves to a data state). In a data state, the automaton checks if the current data value and the configuration of the registers satisfy a condition, and if they do, moves to a word state and updates some of the registers with the read data value.

Given a data word $w = d_0 a_0 d_1 a_1 \ldots a_{n-1} d_n$, where each d_i is a data value and each a_l is a letter, a configuration of $\mathcal{A}$ on w is a tuple (j, q, λ), where j is the current position of the symbol in w that $\mathcal{A}$ reads, q is the current state and $\lambda \in \mathcal{D}^k$ is the current state of the registers. The initial configuration is $(0, q_0, \lambda_0)$ and any configuration (j, q, λ) with $q \in F$ is a final configuration.

From a configuration $C = (j, q, \lambda)$ we can move to a configuration $C' = (j + 1, q', \lambda')$ if one of the following holds:

- the jth symbol is a letter a, there is a transition $(q, a, q') \in \delta_w$, and $\lambda' = \lambda$; or

- the current symbol is a data value d, and there is a transition $(q, c, I, q') \in \delta_d$ such that $d, \lambda \models c$ and λ' coincides with λ except that the ith component of λ' is set to d whenever $i \in I$.

A data word w is accepted by $\mathcal{A}$ if $\mathcal{A}$ can move from the initial configuration to a final configuration after reading w. We then say that the sequence of configuration forms an *accepting run* for $\mathcal{A}$ on input w. The language of data paths accepted by $\mathcal{A}$ is denoted by $\mathcal{L}(\mathcal{A})$.

Equivalence with RQMs, problem definition

It was shown in [29] that for every RQM e once can construct in polynomial time a register data word automaton $\mathcal{A}_e$ such that $\mathcal{L}(e) = \mathcal{L}(\mathcal{A}_2)$. Let then e_1 and e_2 be RQMs. To show that $e_1 \subseteq e_2$ we can, by Lemma 3.5, show instead that $\mathcal{L}(e_1) \subseteq \mathcal{L}(e_2)$. Moreover, by the aforementioned equivalence with automata, it suffices to show that $\mathcal{L}(\mathcal{A}_{e_1}) \subseteq \mathcal{L}(\mathcal{A}_{e_2})$.

The reminder of the proof is devoted to showing that such decision problem belongs to EXPSPACE, assuming both $\mathcal{A}_1$ and $\mathcal{A}_2$ use only equalities.

A special family of data words

Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two register automata that only use equalities on the conditions, and assume that $\mathcal{L}(\mathcal{A}_1) \not\subseteq \mathcal{L}(\mathcal{A}_2)$. Then there is a data word $w = d_1 a_1 d_2 a_2 \cdots a_n d_{n+1}$ that belongs to $\mathcal{L}(\mathcal{A}_1)$ but it does not belong to $\mathcal{L}(\mathcal{A}_2)$. Further, there is an accepting *run* τ that associates to each data value d_i in w a change of configuration, going from a configuration of the form $(2i-1, q, \lambda)$ to one of the form $(2i, q', \lambda')$.

Set $w^1 = w$ and $\tau^1 = \tau$. Starting from $i = 2$ up to $i = n+1$, we repeatedly perform the following operations on w^i.

Let w^{i-1} and τ^{i-1} be the resulting word and accepting run after performing the $i-1$-th operation, and assume that τ_{i-1} changes from a configuration $(2i-1, q, \lambda)$ to $(2i, q', \lambda')$. If all data values in λ are also in λ', then let $w^i = w^{i-1}$ and $\tau^i = \tau^{i-1}$. Otherwise, assume that $d^1, \ldots, d^k$ are in λ but not in λ'. Then let $p^1, \ldots, p^k$ be fresh, new data values. Construct w^i as follows. For each $j = 1, \ldots, k$, replace all appearances of d^j in w_{i-1}, only after position $2i-2$ of w^{i-1}, for the data value p^j. Moreover, construct τ^i by replacing as well $d^1, \ldots, d^k$ for $p^1, \ldots, p^k$ in all the register values of the remaining configurations, from position $2i-1$ onwards.

Given arbitrary automaton $\mathcal{A}_1$, word $w \in \mathcal{L}(()\mathcal{A}_1)$ and run τ witnessing the acceptance of w, let us denote by $u_{w,\tau}$ the resulting word w^{n+1} after performing all transformations above, and by $\sigma_{w,\tau}$ the resulting run τ^{n+1}. Note that the constructed run remains a valid run, so that $\mathcal{A}_1$ accepts as well the word $u_{w,\tau}$. Moreover, the following can be shown about $u_{w,\tau}$ (the proof follows by construction):

CLAIM 8.2. *Assume that there are positions j_1 and j_2 of $u_{w,\tau}$ such that both j_1 and j_2 contain the same data value. Then such data value is present in at least one register in all configurations of $\sigma_{w,\tau}$ starting from position j_1 and ending in position j_2.*

Moreover, it is easy to see:

CLAIM 8.3. *If any other automaton $\mathcal{A}_2$ does not accept w, then it does not accept $u_{w,\tau}$.*

This follows simply because we are only using automata with equalities, and our transformation actually introduce additional inequalities on the data values of words. From the above claims we obtain the following

LEMMA 8.4. *Given automata $\mathcal{A}_1$ and $\mathcal{A}_2$, we have that $\mathcal{L}(()\mathcal{A}_1) \subseteq \mathcal{L}(()\mathcal{A}_2)$ if and only if there is a word $w \in \mathcal{L}(()\mathcal{A}_1)$, accepted by run τ, and such that $u_{w,\tau}$ belongs to $\mathcal{L}(()\mathcal{A}_1)$ but does not belong to $\mathcal{L}(()\mathcal{A}_2)$*

All that remains now is to show that the existence of such a word can be decided in EXPSPACE.

Let now $\mathcal{A}_1 = (Q_1, q_1^0, F_1, \lambda_1^0, \delta_1)$ and $\mathcal{A}_1 = (Q_2, q_2^0, F_2, \lambda_2^0, \delta_2)$. furthermore, assume that $REG(\lambda_1)$ and $REG(\lambda_2)$ are all possible valuations of registers in $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively, using elements from $\mathcal{D}$ (obviously these are infinite sets).

Consider the following transition system. Its states are $Q_1 \times REG(\lambda_1) 2^{Q_2 \times REG(\lambda_2)}$. The initial state is $(q_1^0, \lambda_1^0), \{(q_2^0, \lambda_2^0)\}$, the set of final states are all those states that contain a state in F_1 and does not contain any state in F_2 (i.e. if at any point we are in a final state, we know that a given word is accepted by $\mathcal{A}_1$ but it is not accepted by $\mathcal{A}_2$)

The transition is defined as follows: there is a transition between state $(q_1, \lambda_1), \{(q_2^1, \lambda_2^1), \ldots, (q_2^n, \lambda_2^n)\}$ and state $(q_1', \lambda_1'), \{(q'^1_2, \lambda'^1_2), \ldots, (q'^m_2, \lambda'^m_2)\}$ by letter a or data value d if one can go from (q_1, λ_1) to (q_1', λ_1') using δ_1 over a or d, and $\{(q'^1_2, \lambda'^1_2), \ldots, (q'^m_2, \lambda'^m_2)\}$ is the set of all states that are reachable from any state in $\{(q_2^1, \lambda_2^1), \ldots, (q_2^n, \lambda_2^n)\}$, using δ_2 and a or d.

Now, obviously the size of this transition system is infinite. However, we proceed as follows.

We guess, symbol by symbol, the word $u_{w,\tau}$ and its run $\sigma_{w,\tau}$, and only pick those moves in the transition system where q_1 and λ_1 move as in $\sigma_{w,\tau}$. Then by the properties of $u_{w,\tau}$ and $\sigma_{w,\tau}$ we know that any state $(q_1, \lambda_1), \{(q_2^1, \lambda_2^1), \ldots, (q_2^n, \lambda_2^n)\}$

can be simplified into a state in which all values in $\lambda_2^1, \ldots, \lambda_2^n$ that are not in λ_1 are mapped to a single fresh value d. This is because such data values will never appear again in $u_{w,\tau}$, and thus from the equality point it is just as good as any data value which is different to all the remaining values in $u_{w,\tau}$.

But we can do even better, as here it suffices to store only the equivalence classes of the registers, i.e., whether the registers store, at any given point, the same data value as in other register, or a different one. If the next symbol we are guessing corresponds to a data value that was in one of the registers of λ_1, then we guess, instead of the particular data value, the following information "the incoming data value is the one stored in register x". The system then updates the equivalence classes according to the registers. If, on the contrary, the incoming data value is a data value different from all λ_1, we just guess "the incoming data value is not stored in any register", and then updates the information as before.

Thus, for our simulation of $\mathcal{A}_1$ it suffices to store, at any given point, the equivalence class formed by the registers in $\mathcal{A}_1$, and to simulate all possible runs of $\mathcal{A}_2$ we need to store, besides the equivalence classes of its registers, a pointer indicating whether it is storing a value also stored in a register of $\mathcal{A}_1$, or whether it is storing a data value not currently stored in $\mathcal{A}_1$ (that will never show up again in our word). This amounts to a total of $Q_1 \times 2^{|\mathcal{A}_1|} \times 2^{Q_2 \times 2^{|\mathcal{A}_2| \times |\mathcal{A}_1|}}$ states, which is doubly exponential in $\mathcal{A}_1$ and $\mathcal{A}_2$. We can therefore decide whether there is a valid run fo this system (that ends in a final state) using a standard on-the-fly EXPSPACE algorithm.

Hardness. The proof of EXPSPACE-hardness is by reduction from the complement of the acceptance problem of a Turing machine.

Let L be a language that belongs to EXPSPACE over some alphabet Γ, $\mathcal{M}$ be a deterministic Turing machine that decides L in EXPSPACE, and w be a word (plain, without data values) over Γ. Next we show how to construct RQDs e' and e (in polynomial time in the size of $\mathcal{M}$ and w) such that $\mathcal{L}(e') \subseteq \mathcal{L}(e)$ if and only if $\mathcal{M}$ does not accept the input w. By Proposition 3.5 this is enough for the proof of the hardness.

Let $\mathcal{M} = (Q, \Gamma, q_0, \{q_f\}, \delta)$, where $Q = \{q_0, \ldots, q_f\}$ is the set of states, Γ is the tape alphabet, containing the distinguished blank symbol B, q_0 and q_m are the unique initial and final states, and $\delta : (Q \setminus \{q_f\}) \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function. Notice, that without loss of generality we assume that no transition is defined on the unique final state q_f. Since $\mathcal{M}$ decides L in EXPSPACE, there exists a polynomial P (which does not depend on w) such that $\mathcal{M}$ decides w using space 2^n, where $n = P(|w|)$. Let also $w = a_0 a_1 \cdots a_k$.

In the following find convenient to introduce the following abusing of notation. For an alphabet $\Delta = \{b_1, \ldots, b_m\}$ of symbols, we denote by the same Δ the regular expression $(b_1 \cup \cdots \cup b_m)$.

Let $\Sigma = \{\#, \&, \%, \triangle\} \cup \Gamma \cup (\Gamma \times Q)$ be the alphabet of the constructing expressions e' and e.

Let $\langle i \rangle$ denote the binary representation of the number i as a data word on n labels $\#$ such that its data values represent the string representation of i as a binary number. That is, the data word $d_n \# d_{n-1} \# \ldots \# d_1$ such that $d_n d_{n-1} d_1$ is precisely the string representation of i as a binary number. For example, $\langle 0 \rangle$ is the data word $(0\#)^{n-1}0$, and $\langle 2 \rangle$ is the data word $(0\#)^{n-2}1\#0$.

We represent configurations of the Turing machine by data words satisfying

$$\langle 0 \rangle \, (\Gamma \cup (\Gamma \times Q))\, d \quad \& \quad \langle 1 \rangle \, (\Gamma \cup (\Gamma \times Q))\, d \quad \& \quad \langle 2 \rangle \, (\Gamma \cup (\Gamma \times Q))\, d \quad \& \quad \ldots \quad \langle 2^n - 1 \rangle \, (\Gamma \cup (\Gamma \times Q))\, d \,\&\, d \,\%\, d, \quad (4)$$

where d stands for any data value. Intuitively, the words $\langle 0 \rangle$, $\langle 1 \rangle$, $\langle 2 \rangle$, $\langle 2^n - 1 \rangle$ indicate each of the 2^n cells of $\mathcal{M}$, and the symbol following such a word represents either the content of the cell (which means that the head does not point here), or the content of the cell plus the state of $\mathcal{M}$ (if $\mathcal{M}$ is pointing at that particular cell at a given point of the computation).

Since every configuration of $\mathcal{M}$ can be represented as a data word of form (4), a run of $\mathcal{M}$ on the input w can be seen as a sequence (i.e. concatenation) of words of form (4).

The idea of the reduction is the following. The expression e' is such that it accepts all data words in each of which every data value is equal to one of the first two data values of the word. Without loss of generality we can then denote the first data value of each of these words by 0 and the second data value by 1. In turn, the expression e shall represent all those words that belong to $\mathcal{L}(e')$ that are either not valid concatenations of words of form (4), or that the sequence of configurations is not a valid run of $\mathcal{M}$ on input w (in both cases, followed by some initialisation). This way, if there is a valid run for $\mathcal{M}$ on w, we have that there is a data word in $\mathcal{L}(e')$ that is not in $\mathcal{L}(e)$, i.e. $\mathcal{L}(e') \nsubseteq \mathcal{L}(e)$.

Formally, the first of these expressions e' is defined as following:

$$e' = \downarrow x.\triangle \downarrow y.(\triangle[x^=] \cup \triangle[y^=])\big(\Sigma[x^=] \cup \Sigma[y^=]\big)^*.$$

We split the definition of the second expression into six parts $e = e^0 \cup e^1 \cup e^2 \cup e^3 \cup e^4 \cup e^5$, such that

- e^0 describes all words that use a single data value (instead of two);
- e^1 describes all data words that are not concatenations of words of form (4);

- e^2 describes all words that, even if they are concatenations of words of form (4), some of them do not represent valid configurations for $\mathcal{M}$;
- e^3 describes words in which the first configuration does not correctly describe the initial configuration of $\mathcal{M}$ on input w;
- e^4 describes those words in which the last sub word of form (4) does not represent an accepting configuration of $\mathcal{M}$;
- e^5 describes words that contain two consecutive sub words of form (4) that represent configurations for $\mathcal{M}$ which, however, do not agree on δ.

Expression e^0 is straightforward to define. Next we give the remaining ones.

Expression e^1. Most of this expression is not really related to data values, but instead can be defined by an NFA in a standard way (see [4] Theorem 6). The only interesting part is the one which accepts all words with a "configuration" in which "cells" are concatenated not in the only proper order, from $\langle 0\rangle$ to $\langle 2^n-1\rangle$. To do this we include in e^1 the a disjunction of the following expressions:

- the expressions

$$\downarrow x.\triangle\downarrow y.\triangle\, \Sigma^* \,(\#[x^=])^n (\Sigma\setminus\{\%\})^* \,(\#[x^=])^n\, \Sigma^*,$$
$$\downarrow x.\triangle\downarrow y.\triangle\, \Sigma^* \,(\#[y^=])^n (\Sigma\setminus\{\%\})^* \,(\#[y^=])^n\, \Sigma^*,$$

  which look for two words of form $\langle 0\rangle$ within one configuration, and likewise for $\langle 2^n-1\rangle$;
- the expressions

$$\downarrow x.\triangle\downarrow y.\triangle\, \Sigma^*\, \%\, (\#[x^=])^i\, \#[y^=]\, \Sigma^*, \qquad \text{for each } 0\le i\le n-1,$$
$$\downarrow x.\triangle\downarrow y.\triangle\, \Sigma^*\, \#[x^=]\, (\#[y^=])^i\, (\Gamma\cup(\Gamma\times Q))\, \%\, \Sigma^*, \qquad \text{for each } 0\le i\le n-1,$$

  which look for a configuration starting with something different from $\langle 0\rangle$, and likewise ending with something different from $\langle 2^n-1\rangle$;
- the expression

$$\downarrow x.\triangle\downarrow y.\triangle\, \Sigma^*\, \#^{n-1}\, \#[x^=]\, (\Gamma\cup(\Gamma\times Q))\, \&\, \#^{n-1}\, \#[x^=]\, \Sigma^*,$$

  looking for a configuration where an even number follows with another even number;
- the expressions

$$\downarrow x.\triangle\downarrow y.\triangle\, \Sigma^*\, \#^i\, \#[x^=]\, \#^{n-i-2}\, \#[x^=]\, (\Gamma\cup(\Gamma\times Q))\, \&\, \#^i\, \#[y^=]\, \#^{n-i-1}\, \Sigma^*, \qquad \text{for each } 0\le i\le n-2,$$
$$\downarrow x.\triangle\downarrow y.\triangle\, \Sigma^*\, \#^i\, \#[y^=]\, \#^{n-i-2}\, \#[x^=]\, (\Gamma\cup(\Gamma\times Q))\, \&\, \#^i\, \#[x^=]\, \#^{n-i-1}\, \Sigma^*, \qquad \text{for each } 0\le i\le n-2,$$

  looking for a configuration where an even number follows with a number where some of the digits are different from the onces in the previous number (except the last).

Note that last 2 cases cover all configurations in which even position numbers are not followed by their successors. It is also possible, but rather cumbersome and lengthy, to define expressions which cover the even – odd cases. We omit such definition, and refer the reader to [4] for very similar constructions.

Expression e^2. Similarly to the next expressions e^3 and e^4, it can be described with standard NFA's. In particular, e^2 is the union of expressions stating the following:

- between two symbols $\%$ there is no symbol in $(\Gamma\times Q)$, which means that in some configuration the machine dos not point to any cell;
- between two neighbouring symbols $\%$ there are two symbols in $(\Gamma\times Q)$, which means that the machine is pointing at two cells.

Expression e^3. It is the union of expressions stating the following:

- the first configuration does not contain the initial state in the first position of the tape, reading the first symbol of the input;
- the following $k-1$ cells do not contain the remainder of the input;
- any of the remaining cells does not contain the blank symbol.

Expression e^4. It can be dfined in the similar way as e^3.

Expression e^5. It is defined as the union of the following expressions:

- a cell not pointed by the head changed its content from one configuration to the subsequent one:

$$\bigcup_{a\in\Gamma} \downarrow x.\triangle\downarrow y.\triangle\,\Sigma^*\,\#\downarrow x_1.\#\,s\downarrow x_{n-1}.\#\downarrow x_n.a\,(\Sigma\setminus\{\%\})^*\,\%$$
$$(\Sigma\setminus\{\%\})^*\,\#[x_1^=]\,\#[x_2^=]\,s\#[x_n^=]\,\big((\Gamma\setminus\{a\})\cup((\Gamma\setminus\{a\})\times Q)\big)\,\Sigma^*;$$

- a configuration which is not final features a pair in $\Gamma\times Q$ for which no transition is defined

$$\bigcup_{\{(a,q)\mid\delta(q,a)\text{ is not defined}\}} \Sigma^*\,(a,q)\,\Sigma^*\,\%\,\Sigma^+;$$

- the change of state does not agree with δ:

$$\bigcup_{\{(a,q)\mid\delta(q,a)=(a',q',\{L,R\})\}} \Sigma^*\,(a,q)\,(\Sigma\setminus\{\%\})^*\,\%\,(\Sigma\setminus\{\%\})^*\,\big(\Gamma\times(Q\setminus\{q'\})\big)\,\Sigma^*;$$

- the symbol written in a given step does not agree with δ:

$$\bigcup_{\{(a,q)\mid\delta(q,a)=(a',q',\{L,R\})\}} \downarrow x.\triangle\downarrow y.\triangle\,\Sigma^*\,\#\downarrow x_1.\#\,s\downarrow x_{n-1}.\#\downarrow x_n.(a,q)\,(\Sigma\setminus\{\%\})^*\,\%$$
$$(\Sigma\setminus\{\%\})^*\#[x_1^=]\,\#[x_2^=]\,s\#[x_n^=]\,(\Gamma\setminus\{a'\})\,\Sigma^*;$$

- the movement of the head does not agree with δ:

$$\bigcup_{\{(a,q)\mid\delta(q,a)=(a',q',R)\}} \downarrow x.\triangle\downarrow y.\triangle\,\Sigma^*\,\#\downarrow x_1.\#\,s\downarrow x_{n-1}.\#\downarrow x_n.(a,q)\,(\Sigma\setminus\{\%\})^*\,\%$$
$$(\Sigma\setminus\{\%\})^*\,\#[x_1^=]\,\#[x_2^=]\,s\#[x_n^=]\,a'\,\&\,(\varepsilon\cup\big(\#^n\,\Gamma\,(\Sigma\setminus\{\%\})^*\big))\,\%\,\Sigma^*,$$

$$\bigcup_{\{(a,q)\mid\delta(q,a)=(a',q',L)\}} \downarrow x.\triangle\downarrow y.\triangle\,\Sigma^*\,\#\downarrow x_1.\#\,s\downarrow x_{n-1}.\#\downarrow x_n.(a,q)\,(\Sigma\setminus\{\%\})^*\,\%$$
$$(\varepsilon\cup\big((\Sigma\setminus\{\%\})^*\,\#^n\,\Gamma\,\&\big))\,\#[x_1^=]\,\#[x_2^=]\,s\#[x_n^=]\,\Sigma^*.$$

With these definitions in hand, it is now straightforward to show that $\mathcal{L}(e')\subseteq\mathcal{L}(e)$ if and only if $\mathcal{M}$ does not accept on input w. This finishes the proof of the EXPSPACE lower bound. □

Proofs for Section 4

PROPOSITION 4.3. *Given two RQDs e_1 and e_2, it holds that $e_1\subseteq e_2$ iff $\mathcal{L}(e_1)\subseteq\mathcal{L}(e_2)$.*

PROOF. Proof of the proposition is analogous to the proof of Proposition 3.5. but this time using the fact [29] that for RQDs a pair of nodes (u,v) belongs to $[\![e]\!]^G$ if and only if there is a path from u to v such that its corresponding data word belongs to $\mathcal{L}(e)$. □

THEOREM 4.4. *The problem* CONTAINMENT *(*RQDs*) is undecidable.*

PROOF. Next we will prove a stronger result that the universality problem for RQDs, defined below, is undecidable. Let $\Sigma[\mathcal{D}]^*$ denote the set of all data words over the alphabet Σ and set of data values $\mathcal{D}$.

UNIVERSALITY OF RQDs	
Input:	An RQD e.
Qestion:	Does $\mathcal{L}(e)=\Sigma[\mathcal{D}]^*$?

The undecidability of this problem immediately implies that given two RQDs e_1 and e_2, checking whether $L(e_1) \subseteq L(e_2)$ is undecidable. The latter then implies undecidability of query containment over graphs by Proposition 4.3.

The proof of undecidability of universality problem for RQDs is similar to the proof of the universality of register automata in [33]. The reduction is from *Post correspondence problem (PCP)*, which is well-known to be undecidable.

An *instance of PCP* is a set of pairs of words

$$\{(u_1, v_1), \ldots, (u_n, v_n)\}, \tag{5}$$

over a finite alphabet Γ. A *solution* for an instance I is a sequence $k_1, \ldots, k_m$ of numbers from $\{1, \ldots, n\}$ such that $u_{k_1} \cdots u_{k_m} = v_{k_1} \cdots v_{k_m}$. The question is whether an instance has a solution.

Throughout the reduction we will use the following notation for every data word $w = d_1 a_1 d_2 \ldots a_{k-1} d_k$. Let REV$(w)$ be the reversal of w, that is REV$(w) = d_k a_{k-1} \ldots d_2 a_1 d_1$. Also, let $\mathsf{Proj}(w)$ be its projection to the labels, i.e. the word $a_1 \ldots a_{k-1}$.

Let $\$, \#$ be two special symbols not in Γ, let $\Sigma' = \Gamma \cup \{\$, \#\}$, and let $\Sigma = \Gamma \cup \{\$\}$. A solution $k_1, \ldots, k_m$ of a PCP instance I of the form (5) can be encoded as a data word $w_1\#\text{REV}(w_2)$ over Σ, where

$$w_1 = \\ 0\ \$c_1\ a_1 d_1 \cdots a_{\ell_1} d_{\ell_1}\ \ \$c_2\ a_{\ell_1+1} d_{\ell_1+1} \cdots a_{\ell_1+\ell_2} d_{\ell_1+\ell_2} \cdots\cdots\ \$c_m\ a_{\ell_1+\cdots+\ell_{m-1}+1} d_{\ell_1+\cdots+\ell_{m-1}+1} \cdots a_{\ell_1+\cdots+\ell_m} d_{\ell_1+\cdots+\ell_m},$$

$$w_2 = \\ 0\ \$g_1\ b_1 f_1 \cdots b_{\ell_1} f_{\ell_1}\ \ \$g_2\ b_{\ell_1+1} f_{\ell_1+1} \cdots b_{\ell_1+\ell_2} f_{\ell_1+\ell_2}\ \cdots\cdots\ \$g_m\ b_{\ell_1+\cdots+\ell_{m-1}+1} f_{\ell_1+\cdots+\ell_{m-1}+1} \cdots b_{\ell_1+\cdots+\ell_m} f_{\ell_1+\cdots+\ell_m},$$

such that a's and b's are labels from Σ, c's, g's, d's, f's, and 0 are data values, and, for a shortcut $\ell = \ell_1 + \cdots + \ell_m$, the following conditions hold:

(C1) the symbol $\#$ appears only once;

(C2) $\mathsf{Proj}(w_1) \in (\$u_1 \cup \cdots \cup \$u_n)^*$;

(C3) $\mathsf{Proj}(w_2) \in (\$v_1 \cup \cdots \cup \$v_n)^*$;

(C4) the data values c_i's and d_i's are pairwise different;

(C5) the data values g_i's and f_i's are pairwise different;

(C6) $c_1 = g_1$ and $c_m = g_m$;

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

(C8) for each $i, j \in \{1, \ldots, m-1\}$ if $c_i = g_j$ then $c_{i+1} = g_{j+1}$;

(C9) for each $i, j \in \{1, \ldots, \ell-1\}$, if $d_i = f_j$ then $d_{i+1} = f_{j+1}$;

(C10) for each $i, j \in \{1, \ldots, \ell\}$, if $d_i = f_j$, then $a_i = b_j$;

(C11) for each $i, j \in \{1, \ldots, m\}$, if $c_i = g_j$, then $(a_{\ell_1+\ldots+\ell_{i-1}+1} \cdots a_{\ell_1+\ldots+\ell_i}, b_{\ell_1+\ldots+\ell_{j-1}+1} \cdots b_{\ell_1+\ldots+\ell_j}) \in I$.

Note that e.g. Conditions (C4–C6, C8) forces the sequence of c's in w_1 to be equal to the sequence of g's in w_2.

It is straightforward to show that there exists a solution to the PCP instance I if and only if there exists a data word of the form $w_1\#\text{REV}(w_2)$ over Σ' that satisfies Conditions (C1–C11) above. The word w_1 is meant to encode the u-part of I and w_2 the v-part. The idea is that the equality $c_i = g_i$ codes a position k_i in a solution by a unique data value, and in (C11) it is checked that the pair on this position belongs to I. Also, d's and f's code the actual pairs (u_i, v_i) in I and since we check that d's equal f's in Conditions (C4–C9) and that the letter after each d equals the corresponding one before the appropriate f in Condition (C10). Note that we require the word w_2 to be reversed in order to nest equality tests according to the semantics of RQDs.

We now construct an RQD e over Σ' that accepts a data word w such that it is either not of the form $w_1\#\text{REV}(w_2)$, or at least one of the Conditions (C1–C11) above is not satisfied. Thus, if e is universal (i.e. accepts all data words) then in particular there is no data word coding a solution to the PCP instance, and, hence there is no solution by itself. The RQD e is obtained by taking the union of the following, using the usual shortcut Δ for the expression $b_1 \cup \ldots \cup b_p$ over any alphabet $\Delta = \{b_1, \ldots, b_p\}$:

- RQDs recognising the negations of Conditions (C1–C3), which can be written as standard regular expressions without equality tests;
- the RQD

$$\Big(\Sigma^*\$(\Gamma\Sigma^*\$)_=\Sigma^* \quad \cup \quad \Sigma^*\$\Gamma\Gamma^*(\Sigma^*\Gamma)_=\Big)\#\Sigma^*,$$

  which recognises the negation of (C4); here the left part of $\cup$ finds equal c's, while the right one finds equal d's; note that for equal ds we take care that we don't incidentally compare with some c;

- an RQD which recognises the negation of (C5), which is very similar to the previous one, but takes into account that w_2 is reversed;
- the RQD

$$\$(\Sigma^*)_{\neq}\$ \quad \cup \quad \Sigma^*\$(\Gamma^*\#\Gamma^*)_{\neq}\$\Sigma^*,$$

  which recognises the negation of (C6); note, that here we use the fact that w_2 is reversed, so in particular g_1 appears as the second last data value (and right before the final \$), which is covered by the left disjunct; similarly c_m is the value after the last \$ in w_1, so after that we can only advance by means of Γ before reaching $\#$ and then we proceed in w_2 to the first \$ in front of which g_m is located;
- an RQD which recognises the negation of (C7), which is very similar to the previous one;
- the RQD

$$\Sigma^*\$(\Gamma^*\$(\Sigma^*\#\Sigma^*\$)_{\neq}\Gamma^*\$)_{=}\Sigma^*,$$

  which recognises the negation of (C8);
- RQDs which recognise the negation of (C9–11), which are very similar to the previous one.

It is straightforward to see that the PCP instance I has no solution if and only if $\mathcal{L}(e) = \Sigma[\mathcal{D}]^*$. This concludes our proof of Theorem 4.4. □

THEOREM 4.5. *The problem* CONTAINMENT *(*positive RQDs*) is* PSPACE*-complete.*

PROOF. The hardness immediately follows from the PSPACE-completeness of the containment problem of RPQs, so next we concentrate on the algorithm. Before its description we introduce a monoid Λ which domain is a set of pairs of nonnegative numbers and which operation $\circ$ is defined as follows:

$$\langle \ell_1', \ell_2' \rangle \circ \langle \ell_1'', \ell_2'' \rangle = \begin{cases} \langle \ell_1', \ell_2' - \ell_1'' + \ell_2'' \rangle, & \text{if } \ell_2' \geq \ell_1'', \\ \langle \ell_1' - \ell_2' + \ell_1'', \ell_2'' \rangle, & \text{otherwise.} \end{cases}$$

An *NFA with positive data tests* is a tuple $(Q, \Sigma, \delta, q_0, q_f, \gamma)$, where

- $\langle Q, \Sigma, \delta, q_0, q_f \rangle$ is a usual NFA without ε-transitions, such that every node is on a path from q_0 to q_f,
- γ is a partial function $\gamma : Q \times Q \rightharpoonup \Lambda$, defined for all pairs q_1, q_2 for which there exists a transition from q_1 to q_2 by some symbol in Σ, and
- for any loop along transitions in this automata, as well as for any path from q_0 to q_f the composition of values of γ is $\langle 0, 0 \rangle$.

Such an NFA is essentially a register automata restricted by a special policy for manipulation of registers:

1. only tests for equality are allowed,
2. the registers are arranged in a stack,
3. data values can be stored only in the currently unused register just above the top of the stack (i.e., pushed), and
4. (positive) comparisons of current data value can be performed with only the value stored in the register on top of the stack, and, moreover, after such a comparison the stored value is lost and the register becomes unused (i.e., the value is popped).

Having this policy, we do not need to name the registers, and the function γ represents the number of values which are popped from the stack as the first component, and the number of times the current data value is pushed as the second component. Note, that the last requirement of NFA with positive data tests guarantees that the number of registers used (i.e. the size of the stack) is bounded by a number which does not depend on the particular input and run. The semantics of NFA with positive data tests is inherited from register automata.

Given an RQD which last symbol is from Σ, the corresponding NFA with positive data tests is as following. The NFA part is as the standard transformation from the regular expression to NFA (without ε-transitions). For each transition in this NFA there is a corresponding sequence of openings and closings of data values equality checks in the RQD, which is done before reading the symbol; however, without loss of generality we may assume, that there are no openings after any closings, i.e. this sequence can be represented as a pair of nonnegative numbers. This pair will be the value of γ for the states from the transition. Note, that such a pair is unique for any two states, regardless which valid transition symbol we take for these states.

The transformation above is straightforward and can be done in polynomial time. Moreover, the language $\mathcal{L}(A)$ of the NFA with positive data tests corresponding to an RQD e is the same as the language $\mathcal{L}(e)$. The size of a stack required for such an automata (which does not depend on input, as noted above) equals the maximal depth of nesting of $()_=$ in e.

Let e' and e be positive RQDs. Without loss of generality we assume that the last symbol of each of them is from Σ. Next we describe a PSPACE algorithm which decides whether $\mathcal{L}(A') \subseteq \mathcal{L}(A)$, where A' and A are the NFAs with positive data tests which correspond to e' and e. By the observation above and Proposition 4.3 it is enough for the proof of the theorem.

In the algorithm the following data structures are used.

- A state q' in A'; it is used to represent the current state which moves on each step non-deterministically according to a transition of A'.
- A stack $\mathcal{P}$ of positive natural numbers; the sum of all numbers in this stack always equals to the nesting depth of q', so we can always use only polynomial space to store it; we denote $|\mathcal{P}|$ the number of positions in $\mathcal{P}$; we assume that the positions in the stack are enumerated from 1 and refer to those numbers by just *positions*; this stack essentially splits all the currently open equality checks in the automata A' into consecutive groups for each of which we know that the opening data values of all the checks in this group are the same.
- A set $\mathcal{G}$ of quadruples of the form (q_1, q_2, n, λ), where $q_1, q_2 \in Q$, n is a position in $\mathcal{P}$, and λ is a pair from Λ; during the run of the algorithm the following always hold for each (q_1, q_2, n, λ) in $\mathcal{G}$:

  if $n = 1$ then $q_1 = q_0$,

  if $1 < n \leq |\mathcal{P}|$ then there exists a quadruple $(q_1', q_1, n-1, \lambda')$ in $\mathcal{G}$ for some q_1' and λ';

  as an exception, it will be convenient to have the quadruple $(*, q_0, 0, *)$ (for a special symbol $*$, which is however never used) in $\mathcal{G}$ during all the run of the algorithm; this set represents the history of reactions in A to the transitions in A' which open equality checks.
- A set $\mathcal{F}$ of pairs of states from Q; this set represents the reactions in A inside the current equality check in A': the first component stores the second component of the "parent" quadruple in $\mathcal{G}$ (i.e., one of the quadruples for which the third component is $|\mathcal{P}|$), and the second is (one of) the current states in A.

The stack $\mathcal{P}$ and the set $\mathcal{G}$ are highly related, so we often consider them as a pair $(\mathcal{P}, \mathcal{G})$. Given a positive number m, which is less than the sum of all numbers in $\mathcal{P}$, a *trace* of depth m in $(\mathcal{P}, \mathcal{G})$ is a tuple $(q_1, q_2, n, k, \lambda)$ where $q_1, q_2 \in Q$, n is a position in $\mathcal{P}$, k is a number, and $\lambda \in \Lambda$, such that

- n is the position in $\mathcal{P}$ such that the sum s of all numbers in the positions greater than n is less than m, but if we add the number r in the position n, it is greater or equal;
- $k = s + r - m$;
- there is a sequence $(q_1^i, q_2^i, i, \lambda^i)$, $n \leq i \leq |\mathcal{P}|$, of quadruples from $\mathcal{G}$ such that $q_1^n = q_1$ and $q_2^{|\mathcal{P}|} = q_2$;
- $\lambda = \lambda^n \circ \ldots \circ \lambda^{|\mathcal{P}|}$.

Note, that even if the number of sequences above can be exponential in the size of $\mathcal{G}$, the number of traces is polynomial. Also, all the traces of depth m in $(\mathcal{P}, \mathcal{G})$ depend only on $\mathcal{P}$, so they have the same n and k. If $k = 0$ we say that the *level of depth* m in $\mathcal{P}$ is *exact*. We also define an operation *trim up to depth* m on $(\mathcal{P}, \mathcal{G})$ which is removing all the elements with positions greater or equal than n from $\mathcal{P}$, and all the quadruples from $\mathcal{G}$ which refer to those positions (where n is computed as above).

In turn, we define an operation *flash* of $\mathcal{F}$ which first empties $\mathcal{F}$ and then add there a pair (q_2, q_2) for each $(q_1, q_2, |\mathcal{P}|, \lambda)$ in $\mathcal{G}$.

The algorithm works as follows.

1. Initialize $q' := q_0'$, $\mathcal{P} := \emptyset$, $\mathcal{G} := (*, q_0, 0, *)$, $\mathcal{F} := \{(q_0, q_0)\}$.
2. Repeat until the space is exhausted
   - pick randomly a symbol $a \in \Sigma$ and a state $q'' \in \delta(q', a)$;
   - if $\gamma(q', q'') = \langle 0, 0 \rangle$ then

     form $\mathcal{F}'$ as a set of all pairs (q_1, q_3) such that

       there exists a state q_2 with $(q_1, q_2) \in \mathcal{F}$ and $q_3 \in \delta(q_2, a)$ for which $\gamma(q_2, q_3) = \langle 0, 0 \rangle$,

     $\mathcal{F} := \mathcal{F}'$;
   - else if $\gamma(q', q'') = \langle 0, m_2 \rangle$ then

     append m_2 to $\mathcal{P}$,

     add to $\mathcal{G}$ all quadruples (q_1, q_3, n, λ) such that

there exists a state q_2 for which $(q_1, q_2) \in \mathcal{F}$ and $q_3 \in \delta(q_2, a)$, and
$n = |\mathcal{P}|$, and
$\lambda = \gamma(q_2, q_3)$,
flash $\mathcal{F}$;
- else if $\gamma(q', q'') = \langle m_1, 0 \rangle$ such that the level of depth m_1 in $\mathcal{P}$ is exact then
form $\mathcal{F}'$ as a set of all pairs (q_1, q_4) such that
there exists a trace $(q_1, q_2, n, 0, \lambda)$ in $(\mathcal{P}, \mathcal{G})$ of depth m_1 (for some q_2, λ, and irrelevant number n) and
there exists a state q_3 with $(q_2, q_3) \in \mathcal{F}$ and $q_4 \in \delta(q_3, a)$ for which $\lambda \circ \gamma(q_3, q_4) = \langle 0, 0 \rangle$,
$\mathcal{F} := \mathcal{F}'$;
trim $(\mathcal{P}, \mathcal{G})$ up to depth m_1;
- else let $\gamma(q', q'') = \langle m_1, m_2 \rangle$ and
form $\mathcal{G}'$ as the set of all quadruples (q_1, q_4, n, λ) such that
there exists a trace $(q_1, q_2, n, k, \lambda')$ in $(\mathcal{P}, \mathcal{G})$ of depth m_1 (for some q_2, n, k and λ'), and
there exists a state q_3 with $(q_2, q_3) \in \mathcal{F}$ and $q_4 \in \delta(q_3, a)$,
$\lambda = \lambda' \circ \gamma(q_3, q_4)$,
trim $(\mathcal{P}, \mathcal{G})$ up to depth m_1,
append $k + m_2$ to $\mathcal{P}$,
add $\mathcal{G}'$ to $\mathcal{G}$,
flash $\mathcal{F}$;
- $q' := q''$;
- if $q' = q'_f$ and $(q_0, q_f) \notin \mathcal{F}$ then return **false**.

3. Return **true**.

* □

Proofs for Section 5

PROPOSITION 5.2. *The problem of deciding whether a pair of nodes belongs to $[\![e]\!]^G$ for a 2RQM e and data graph G is* PSPACE*-complete. The same problem is in* PTIME *if e is a 2RQD. If we assume that e is fixed the problem becomes* NLOGSPACE*-comlete.*

PROOF. Take any 2RQM or 2RQD e over Σ and a data graph G. Let $\Sigma' = \Sigma \cup \{a^- : a \in \Sigma\}$ and let $G' = \langle V, E', \rho \rangle$, where V and ρ are as in G, while $E' = E \cup \{(v', a^-, v) : (v, a, v') \in E\}$. Note that we can view e as an ordinary one-way RQM(or RQD) over this extended alphabet. A straightforward induction on expressions shows that $(v, v') \in [\![e]\!]^G$, where e is viewed as an two-way query over Σ, if and only if $(v, v') \in [\![e]\!]^{G'}$, where e is now a (one-way)query over Σ'.

The desired upper bounds now follow from query evaluation algorithms for RQMs and RQDs from [29], since both the alphabet and the graph grow only linearly in size. □

THEOREM 5.3. *The problem* CONTAINMENT *(*positive 2RQMs*) is undecidable.*

PROOF. The proof is by reduction from the problem of non-emptiness of deterministic, stateless 2-way 3-head automata, which is proven to be undecidable in (Yang, L., Dang, Z., and Ibarra, O. H. (2008). On stateless automata and P systems. International Journal of Foundations of Computer Science, 19(05), 1259-1276).

Formally, a *deterministic stateless 2-way 3-head automaton* (or, *DS23A*) over a finite alphabet Γ is given by a transition partial function $\delta : \Sigma \times \Sigma \times \Sigma \rightharpoonup \{-1, 0, 1\}^3$, where $\Sigma = \Gamma \cup \{\vdash, \dashv\}$, the latter symbols assumed not to be in Γ. These automata accept language of words of form $\vdash \sigma \dashv$, with σ a word over Γ. The automaton starts with its 3 heads reading the $\vdash$ symbol of just before σ, moves its heads according to δ (-1 denotes "move one cell back", 0 – "no move", and 1 – "move one cell forward"), and accepts σ if at any step of computation over this word all 3 heads point at the symbol $\dashv$.

Let $\mathcal{A}$ be a DS23A. We now construct 2RQMs e' and e over Σ such that the language of $\mathcal{A}$ is empty if and only if $e' \subseteq e$.

The definition of e' is defined as follows:

$$e' \quad = \quad \vdash \Gamma^* \dashv .$$

As expected, the definition if e is much more intricate. But before it we present a crucial claim.

CLAIM 8.5. *Let e' be the RPQ defined as above, and let e be a 2RQM. Then $e' \subsetneq e$ is and only if there exists a graph G_w corresponding to a data word w with start and end nodes u and v, respectively, such that $(u,v) \in [\![e']\!]^{G_w}$ but $(u,v) \notin [\![e]\!]^{G_w}$.*

PROOF. The if direction is obvious, so we only show the only if direction. Assume then that $e' \subsetneq e$. Then there is a graph G and a pair (u',v') of nodes in G such that $(u',v') \in [\![e']\!]^G$ but $(u',v') \notin [\![e]\!]^G$. Consider a data word w which is a projection of labels and data values of a path in G witnessing e'. Then let us consider the graph G_w corresponding to w, with start and end nodes u and v, respectively. Clearly, $(u,v) \in [\![e']\!]^{G_w}$. Now assume for the sake of contradiction that $(u,v) \in [\![e]\!]^{G_w}$. By examining the definition of 2RQMs one immediately obtains that $(u,v) \in [\![e]\!]^G$, which results in a contradiction. This implies that $(u,v) \notin [\![e]\!]^G$, which was to be shown. □

Next we continue with the definition of e. The idea is the following. Since $\mathcal{A}$ is deterministic, if $\mathcal{A}$ accepts some word σ then there exists a single run that leads to this acceptance. We can take advantage of this determinism, and code with e all computations of $\mathcal{A}$ that end up failing at some point. This way, if there is a data word with a corresponding data graph accepting by e', which is not accepted by e, then the language of $\mathcal{A}$ is nonempty, as $\mathcal{A}$ really accepts this word.

The definition of e is split into three parts as follows:

$$e \quad = \quad e_{\text{eq}} \cup e_{\text{crash}} \cup e_{\text{notdef}}.$$

Intuitively, e_{eq} accepts all graphs corresponding to data words that have two equal data values (data values shall be used as placeholders for the positions of the heads of $\mathcal{A}$, as will be explained shortly); e_{crash} corresponds to words for which the computation of $\mathcal{A}$ crashes, and e_{notdef} corresponds to all words for which the computation of $\mathcal{A}$ ends up in a position that is not defined.

The part e_{eq} is straightforward to define. For definitions of the other parts of e we first need to describe the 2RQM e_{valid}, that simulates the computation of $\mathcal{A}$ on its input.

For each (a,b,c) in Σ^3 for which δ is defined, assume that $\delta(a,b,c) = (t_1,t_2,t_3)$, where each t_i is either -1, 0 or 1. Then let $e_{(a,b,c)}$ be the following expression:

$$(\Sigma^-)^* \vdash \Sigma^*[x_1^=]\, a\, (\Sigma^-)^* \vdash \Sigma^*[x_2^=]\, b\, (\Sigma^-)^* \vdash \Sigma^*[x_3^=]\, c$$
$$(\Sigma^-)^* \vdash \Sigma^*[x_1^=]\, r_1\, (\Sigma^-)^* \vdash \Sigma^*[x_2^=]\, r_2\, (\Sigma^-)^* \vdash \Sigma^*[x_3^=]\, r_3,$$

where, as usual, Σ stands for the union of all symbols in the alphabet Σ, Σ^- stands for the union of inverses of all symbols in Σ, and for each i, $1 \leq i \leq 3$,

$$r_i = \begin{cases} \Sigma^- \downarrow x_i. , & \text{if } t_i = -1, \\ \varepsilon, & \text{if } t_i = 0, \\ \Sigma \downarrow x_i. , & \text{if } t_i = 1. \end{cases}$$

Having this construction in hands, let

$$e_{\text{valid}} \quad = \quad \# \downarrow x_1. \downarrow x_2. \downarrow x_3. \left(\bigcup_{(a,b,c) \text{ s.t. } \delta(a,b,c) \text{ is defined}} e_{(a,b,c)} \right)^* .$$

This expression, so far, describes valid computations, up to some step. In order to make sure that we represent all words not accepted by $\mathcal{A}$, we need to accept all words in which this route of valid computation leads to either a crash (by moving out of the word), or to a transition that is not defined.

Specifically, to describe that a run goes out from the computation space, we define

$$e_{\text{crash}} \quad = \quad e_{\text{valid}} \left(\bigcup_{i=1,2,3} \left(\left((\Sigma^-)^*[x_i^=] \vdash \right) \cup \left(\Sigma^*[x_i^=] \dashv^- \right) \right) \right).$$

Furthermore, for each (a,b,c) such that $\delta(a,b,c)$ is not defined, except $(\dashv,\dashv,\dashv)$ (because this is the final step of an accepting computation), define

$$e_{\neg(a,b,c)} \quad = \quad (\Sigma^-)^* \vdash \Sigma^*[x_1^=]\, a\, (\Sigma^-)^* \vdash \Sigma^*[x_2^=]\, b\, (\Sigma^-)^* \vdash \Sigma^*[x_3^=]\, c,$$

and then

$$e_{\text{notdef}} \quad = \quad e_{\text{valid}} \left(\bigcup_{(a,b,c) \text{ s.t. } \delta(a,b,c) \text{ is not defined, and } (a,b,c) \neq (\dashv,\dashv,\dashv)} e_{\neg(a,b,c)} \right).$$

It is now straightforward to show that the language of $\mathcal{A}$ is nonempty if and only if there exists a graph G_w corresponding to a data word w with start and end nodes u and v, respectively, such that $(u, v) \in [\![e']\!]^{G_w}$ but $(u, v) \notin [\![e]\!]^{G_w}$. Application of Claim 8.5 finishes the proof of the theorem.

Proofs for Section 6

THEOREM 6.2. *The* CONTAINMENT *(*$\mathsf{GXPath}_{\text{reg}}$*) problem is undecidable.*

PROOF. The proof follows the main lines of the proof of undecidability of PDL with extras from [19]. To deduce undecidability we do a reduction from a variant of the tiling problem shown to be undecidable in [23] and [9].

First we define the terminology needed to state the problem precisely.

A *finite set of tiles* is a collection $\mathcal{T} = \{T_1, \dots, T_k\}$ of *square tiles*, together with two *edge* relations $\sim_h$ and $\sim_v$. The fact that $T_i \sim_h T_j$ means that the tile T_j can be placed to the right of the tile T_i in a horizontal row, while $T_i \sim_v T_j$ means that T_i can be placed below T_j in a vertical column.

A tiling of the non-negative grid $\mathbb{N} \times \mathbb{N}$ is a function from $t : \mathbb{N} \times \mathbb{N} \to \mathcal{T}$ such that for all i, j:

- $t(i, j) \sim_h t(i + 1, j)$ and,
- $t(i, j) \sim_v t(i, j + 1)$.

Tilings of integer grid $\mathbb{Z} \times \mathbb{Z}$ are defined analogously. We say that a set of tiles can tile $\mathbb{Z} \times \mathbb{Z}$ periodically if there is a tiling of $\mathbb{Z}_n \times \mathbb{Z}_m$ for some positive integers n and m that can be used to tile the entire grid by repeating this segment both vertically and horizontally. One can imagine this tiling as forming a torus since the bottom row can be "glued" to the top one and the same for left and right edge of this finite grid.

Let now S_{notiling} denote the set of all finite sets of tiles that can *not* tile $\mathbb{N} \times \mathbb{N}$ and let S_{period} be the set of all finite sets of tiles that can tile $\mathbb{Z} \times \mathbb{Z}$ periodically.

To prove undecideability we will use the following fact:

FACT 8.6. *Tiling [23, 9] Sets $S_{notiling}$ and S_{period} are recursively inseparable. In particular there is no* recursive *set S such that $S_{period} \subseteq S$ and $S_{notiling} \cap S = \emptyset$.*

Fix the finite alphabet of edge labels $\Sigma = \{U, D, L, R, a\}$. In what follows U is meant to interpret "up", D "down", L "left" and R "right", while a will be used to code the tiles. Note that we can work with only $\{U, R, a\}$, since we can use U^- instead of D ans R^- instead of L, but we opted for the extended alphabet to make the formulas easier to understand.

Let now $\mathcal{T} = \{T_1, \dots, T_k\}$ be a finite set of tiles. For $i = 1 \dots k$ define $\alpha_i = \langle a^i \cap \varepsilon \rangle$. In what follows α_i is meant to denote the placement of the tile T_i at some position in the grid. E.g. $\langle aaa \cap \varepsilon \rangle$ will denote the placement of the tile T_3 and so on.

We also define the following node formulas of GXPath that will be used throughout the proof. First, for every path formula β we define

$$\texttt{loop}(\beta) := \langle \beta \cap \varepsilon \rangle \wedge \neg \langle \beta \cap \overline{\varepsilon} \rangle.$$

This formula extracts all nodes v from the graph that have an outgoing β path and such that every such path ends at v itself. It is easy to check that for any graph database G:

$$[\![\texttt{loop}(\beta)]\!]^G = \{v \in G : (\exists v') \text{ s.t. } (v, v') \in [\![\beta]\!]^G \text{ and } (\forall v') \text{ If } (v, v') \in [\![\beta]\!]^G \text{ then } v = v'\}.$$

Second, for every path expression β and every node test φ we define the following formula:

$$\texttt{when}(\beta, \varphi) := \neg \langle \beta[\neg \varphi] \rangle.$$

The intended meaning of this node formula is to extract all nodes v from a graph such after every β-path starting in v ends with a node belonging to $[\![\varphi]\!]^G$. Again, it is easy to check that for any graph database G:

$$[\![\texttt{when}(\beta, \varphi)]\!]^G = \{v \in G : (\forall v') \text{ If } (v, v') \in [\![\beta]\!]^G \text{ then } v' \in [\![\varphi]\!]^G\}.$$

Associated with the set of tiles $\mathcal{T}$ we define the formula $\gamma_{\mathcal{T}} = \gamma_1 \wedge \gamma_2$.

To define our formula γ_1 we need to be able to force a "square" at any position in our model, both in a clockwise and in anticlockwise direction. This is done by the means of formula `square` which is defined as the conjunction of the following two formulas:

$$\texttt{clockwise} \;:=\; \texttt{loop}(U \cdot D) \;\wedge\; \texttt{when}(U, \texttt{loop}(R \cdot L)) \;\wedge\; \texttt{when}(U \cdot R, \texttt{loop}(D \cdot U)) \;\wedge\; \texttt{when}(U \cdot R \cdot D, \texttt{loop}(L \cdot R)) \;\wedge\; \texttt{loop}(U \cdot R \cdot D \cdot L)$$

$$\texttt{anticlockwise} \;:=\; \texttt{loop}(R \cdot L) \;\wedge\; \texttt{when}(R, \texttt{loop}(U \cdot D)) \;\wedge\; \texttt{when}(R \cdot U, \texttt{loop}(L \cdot R)) \;\wedge\; \texttt{when}(R \cdot U \cdot L, \texttt{loop}(D \cdot U)) \wedge \texttt{loop}(R \cdot U \cdot L \cdot D)$$

Intuitively `clockwise` allows us to define a square starting at some point in our graph and going "up", then "right", then "down" and finally "left", finishing at the same point. It also forces the point to be able to complete the square whenever it has an outgoing "up" arrow U. Similarly `anticlockwise` forces a square starting with "right" and completing it in an obvious way.

Now γ_1 simply states that we can make a square at any point.

$$\gamma_1 := \texttt{when}(U^*, \texttt{when}(R^*, \texttt{square})).$$

Formula γ_2 is going to be responsible for forcing a tiling and is defined next. First, let

$$\alpha = \bigvee_{i=1\ldots k} \alpha_i \wedge \bigwedge_{i=1\ldots k} (\alpha_i \to \bigwedge_{j \neq i} \neg\alpha_j).$$

Note that α simply states that precisely one α_i is true. Here and in the remainder of the proof we use the node formula $\varphi \to \psi$ as a shorthand for $\neg\varphi \vee \psi$.

Next for each i, define β_i as the disjunction of all the α_j such that $T_i \sim_h T_j$. That is β_i is a disjunction of all the tiles that can be placed to the right of the tile i. Similarly, define β^i to be the disjunction of all α_j such that $T_i \sim_v T_j$.

Now let `tile` be the formula denoting that a tile is placed correctly in the grid. Formally:

$$\texttt{tile} := \alpha \wedge \bigwedge_{i=1\ldots k} (\alpha_i \to (\texttt{when}(R, \beta_i) \wedge \texttt{when}(U, \beta^j))).$$

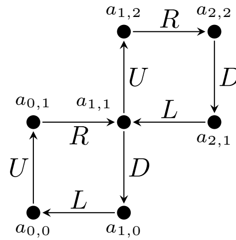
Finally define

$$\gamma_2 := \texttt{when}(U^*, \texttt{when}(R^*, \texttt{tile})).$$

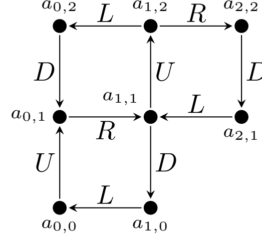
We now show how to deduce the wanted reduction. More formally we show that the set $\{\varphi : \exists G s.t. [\![\varphi]\!]^G \neq \emptyset\}$ contains the set $\{\gamma_{\mathcal{T}} : \mathcal{T} \in \mathcal{S}_{\text{period}}\}$ and is disjoint from $\{\gamma_{\mathcal{T}} : \mathcal{T} \in \mathcal{S}_{\text{notiling}}\}$. Note that Fact 8.6 now implies that $\{\varphi : \exists G s.t. [\![\varphi]\!]^G \neq \emptyset\}$ can not be recursive.

First we show that if $[\![\gamma_{\mathcal{T}}]\!]^G \neq \emptyset$ for some graph G, then $\mathcal{T}$ can tile the positive plane $\mathbb{N} \times \mathbb{N}$. Take any node $a_{0,0} \in [\![\gamma_{\mathcal{T}}]\!]^G$. By γ_1 the proposition `square` has to be true at $a_{0,0}$, so in particular $\texttt{loop}(U \cdot D)$ is true. This means that there is a point which we label $a_{0,1}$ that can be reached from $a_{0,0}$ by an U-labelled edge. (Note that we can also get from $a_{0,1}$ to $a_{0,0}$ by and D-labelled edge.) Now since $\texttt{when}(U, \texttt{loop}(R \cdot L))$ is also true at $a_{0,0}$, there must be a node which we label $a_{1,1}$, reached by an R-labelled edge from $a_{0,1}$ (and with the corresponding L-labelled edge in the other direction). Again, this time using the fact that $\texttt{when}(U \cdot R, \texttt{loop}(D \cdot U))$ is true at $a_{0,0}$, we get a node labelled $a_{1,0}$, connected to $a_{1,1}$ by an D-labelled edge (and with an U-labelled edge connecting it back with $a_{1,1}$). Next, we use the fact that $\texttt{when}(U \cdot R \cdot D, \texttt{loop}(L \cdot R))$ is true at $a_{0,0}$ to get a node $a'_{0,0}$ to the left of $a_{1,0}$. Finally, since $\texttt{loop}(U \cdot R \cdot D \cdot L)$ is true at $a_{0,0}$, it must be that $a'_{0,0} = a_{0,0}$. Again we note that each edge has a dual edge with the appropriate label, connecting the node in reverse direction.

Similarly, since `square` is true at $a_{1,1}$ (as we can reach it from $a_{0,0}$ by traversing U and then R-labelled edge), we can also find points $a_{1,2}, a_{2,2}$ and $a_{2,1}$ in an analogous way. This process is illustrated by the following image (note that we do not claim that nodes $a_{i,j}$ are in fact mutually distinct nodes from our model).



Note now that since `square` is also true at $a_{0,1}$, then $a_{0,1}$ must satisfy `anticlockwise`. Since going R and then U from $a_{0,1}$ takes us to $a_{1,2}$ and since $\texttt{when}(R \cdot U, \texttt{loop}(L \cdot R))$ is true at $a_{0,1}$, there is some node which we label $a_{0,2}$, that is reached by traversing an L-labelled edge from $a_{1,2}$. Note that this also implies that there is an R-labelled edge from $a_{0,2}$ to $a_{1,2}$. Again, since $\texttt{when}(R \cdot U \cdot L, \texttt{loop}(D \cdot U))$ is true at $a_{0,1}$ and $a_{0,2}$ can be reached by $R \cdot U \cdot L$ we have that there is a point $a'_{0,1}$ connected to $a_{0,2}$ by an D-labelled edge (and in the other direction by an U-labelled one). But now since $a_{0,1}$ also satisfies $\texttt{loop}(R \cdot U \cdot L \cdot D)$ and $a'_{0,1}$ is reached from $a_{0,1}$ by a path labelled $R \cdot U \cdot L \cdot D$,we have that $a'_{0,1} = a_{0,1}$. Thus we can draw a square starting in $a_{0,1}$, going in anticlockwise direction. This is illustrated in the following image:



We now note that with each edge there is a corresponding edge in the other direction with the appropriate label (e.g. L and R). To see this observe that in e.d. $a_{0,0}$ we have that $\texttt{loop}(U \cdot D)$is true. This means that there is an U-edge from $a_{0,0}$ to $a_{0,1}$ and also an D-edge from $a_{0,1}$ to $a_{0,0}$ and analogously for all other edges.

In particular there is an R-edge from $a_{0,0}$ to $a_{1,0}$, so we can also complete the clockwise square started at $a_{1,0}$ and continuing through $a_{1,1}$ and $a_{2,1}$. This is done by the means of formula `clockwise`.

It is straightforward to see that this process can be continued for any number of steps, starting from the main diagonal and completing the squares above the diagonal in an anticlockwise direction, while completing the ones below the diagonal in a clockwise direction. Thus we showed that we can force a square grid by our formula.

Define now $t(i, j) = T_l$, where α_l is the unique formula of the form $\langle a^l \cap \varepsilon \rangle$ that is true at any point $a_{i,j}$ by means of γ_2. Note that γ_2 also forces the tiling t to be proper, since the formula `tile` assures that the tile $t(i+1, j)$ and $t(i, j+1)$ can only come from the set of tiles compatible with $t(i, j)$ in the appropriate direction.

Thus we have shown that if formula $\gamma_{\mathcal{T}}$ is satisfiable, then $\mathcal{T}$ can tile the positive plane $\mathbb{N} \times \mathbb{N}$. This implies that the set $\{\varphi : \exists G s.t. [\![\varphi]\!]^G \neq \emptyset\}$ is disjoint from $S_{notiling}$.

On the other hand, suppose that $\mathcal{T} = \{T_1, \ldots, T_k\}$ can tile the plane periodically, that is it can tile the torus $\mathbb{Z}_n \times \mathbb{Z}_m$ for some integers n and m. Let t be the tiling function $t : \mathbb{Z}_n \times \mathbb{Z}_m \to \mathcal{T}$ that witnesses this periodic tiling. We define the graph database G containing at most $(n+1) \cdot (m+1) + (k-2)$ nodes and satisfying $\gamma_{\mathcal{T}}$ as follows.

First, let

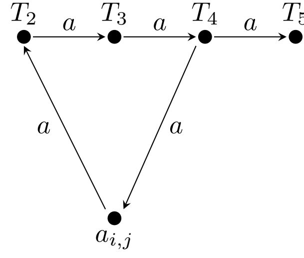
$$V = \{a_{i,j} : i = 1, \ldots, n+1 \text{ and } j = 1, \ldots, m+1\} \cup \{T_2, \ldots, T_k\}.$$

Next add the following edges to our graph:

- For vertical edges:
  - For $i = 1 \ldots n+1$ and $j = 1 \ldots m$ put an U-edge between $a_{i,j}$ and $a_{i,j+1}$ and an D-labelled one in the other direction.
  - For $i = 1 \ldots n+1$ put an U-labelled edge between $a_{i,m+1}$ and $a_{i,1}$ and an D-labelled one in the other direction.
- Analogously for horizontal edges:
  - For $i = 1 \ldots n$ and $j = 1 \ldots m+1$ put an R-edge between $a_{i,j}$ and $a_{i+1,j}$ and an L-labelled one in the other direction.
  - For $j = 1 \ldots m+1$ put an R-labelled edge between $a_{n+1,j}$ and $a_{1,j}$ and an L-labelled one in the other direction.

Also, define $T_2, T_3, \ldots, T_k$ to form an a-labelled chain. That is we add an a-edge between T_i and T_{i+1}, for $i = 2, \ldots k-1$.

Next, for each $a_{i,j}$,where $i \neq n+1$ and $j \neq m+1$ let T_l be the unique tile given by the tiling $t(i, j)$. If $l = 1$ we add an a-edge from $a_{i,j}$ to itself. If $l > 1$ we add an a-labelled edge from $a_{i,j}$ to T_2 and another a-labelled edge from T_l to $a_{i,j}$. This will allow us to satisfy the formula $\alpha_i = \langle a^l \cap \varepsilon \rangle$ as illustrated in the following image.



Finally, for $i = n+1$ and $j \neq m+1$ let $T_l = t(1, j)$ and define the outgoing a-edges from $a_{n+1,j}$ to T_2 and from T_l as above. Similarly, for $i \neq n+1$ and $j = m+1$ do the same for $T_l = t(i, 1)$. Lastly, repeat the procedure for $a_{n+1,m+1}$ and $T_l = t(1, 1)$.

Consider now formula γ_1. Note that we can reach any point by using U and R transitions, so we have to check that $\texttt{square}$ is true at any point. But this is straightforward to check, since our graph G is a simple finite grid that folds onto itself (that is from each point on the edge we can continue in the appropriate direction). The fact that γ_2 is true follows from the fact that t is a periodic tiling. Namely, at any point in the graph G, precisely one α_i is true (note that we require the a-path to loop over the node, so only one such path exists by our construction). After that, any R or U step we take will take us to a node where the appropriate β_j or β^j is true since t is a tiling.

This shows that the set $S = \{\varphi : \exists G s.t. [\![\varphi]\!]^G \neq \emptyset\}$ contains the set $\{\gamma_{\mathcal{T}} : \mathcal{T} \in \mathcal{S}_{period}\}$. As mentioned above, Fact 8.6 implies that the set of all satisfiable GXPath node formulas S, is not recursive.

In particular this implies that query containment for GXPath is not decidable, since the latter would entail recursivity of the set S by simply checking does the containment $[\varphi] \subseteq [\neg\top]$ hold.

Thus we proved that query containment for GXPath is undecidable, even with a fixed alphabet Σ of edge labels.

□

THEOREM 6.5. *The decision problem* CONTAINMENT (GXPath$_{\text{reg}}^{\text{path-pos}}$) *is* EXPTIME-*complete.*

PROOF. To show the upper bound we first prove that the problem of query containment for GXPath$_{\text{reg}}^{\text{path-pos}}$ path formulas can be polynomially reduced to the problem of satisfiability of GXPath$_{\text{reg}}^{\text{path-pos}}$ node formulas. The idea is similar to the one used in [39] to show that the two problems are inter-reducible for XPath queries on trees.

Let α and β be two GXPath$_{\text{reg}}^{\text{path-pos}}$ path formulas and let Γ denote the alphabet of all symbols occurring in α and β plus one additional symbol b. It is straightforward to see that if α is not contained in β, then there is a graph G witnessing this non-containment that uses labels from Γ only. (The idea here is that only labels appearing in α and β are relevant, and all the other labels can be replaced by the new label.)

Let now $\Gamma' := \Gamma \times \{0, 1\}$. That is, Γ' contains copies of each label decorated with either 0 or 1. We define α' as a formula obtained from α by replacing each occurrence of a label a by $(a, 0) \cup (a, 1)$ and likewise for β'. Finally, let $\texttt{out}$ be the formula $\bigcup_{a\in\Gamma}(a, 1)$. We show that α is contained in β if and only if the formula

$$\varphi := \langle\alpha'[\texttt{out}]\rangle \wedge \neg\langle\beta'[\texttt{out}]\rangle$$

is not satisfiable.

Assume first that α is not contained in β. Then there is a graph database G and two nodes $v, v' \in G$ such that $(v, v') \in [\![\alpha]\!]^G$, but $(v, v') \notin [\![\beta]\!]^G$. As mentioned above, we can assume, without the loss of generality, that G uses only labels from Γ. Define now G' to be a Γ' labelled graph where each label a is replaced by $(a, 0)$. In addition, we also add a loop from v' to v' labelled $(b, 1)$. Since v' is the only node with an outgoing edge whose label has second component equal to 1 we get that $v \in [\![\varphi]\!]^{G'}$, as required.

On the other hand, assume that φ is satisfiable. Let G' be any graph such that there is $v \in G'$ with $v \in [\![\varphi]\!]^{G'}$. Let G be a graph obtained from G' be replacing every edge labelled $(a, 0)$ or $(a, 1)$ by a (note that the b-edges can be thrown away, since neither α, nor β can access them). Since $v \in [\![\varphi]\!]^{G'}$, there is some $v' \in G'$ such that $(v, v') \in [\![\alpha'[\texttt{out}]]\!]^{G'}$. It is then straightforward to see that $(v, v') \in [\![\alpha]\!]^G$. On the other hand, if we had that (v, v') is in $[\![\beta]\!]^G$, then we would also get that $(v, v') \in [\![\beta'[\texttt{out}]]\!]^{G'}$ (since v' must have an outgoing edge with second component equal to 1 to satisfy $\alpha'[\texttt{out}]$), which contradicts the fact that $v \in [\![\varphi]\!]^{G'}$. Thus α is not contained in β, as required. (Note that it could still be the case that $v \in [\![\langle\alpha\rangle]\!]^G$ and $v \in [\![\langle\beta\rangle]\!]^G$, but we are interested in binary containment.)

We have thus shown that query containment for $\mathsf{GXPath}^{\text{path-pos}}_{\text{reg}}$ path formulas is polynomially reducible to (un)satisfiability of node formulas of the same language. Using this and the fact that $\mathsf{GXPath}^{\text{path-pos}}_{\text{reg}}$ is contained in Propositional Dynamic Logic (in fact $\mathsf{GXPath}^{\text{path-pos}}_{\text{reg}}$ is the same as PDL without variables) we can use the decision procedure for PDL to solve $\mathsf{GXPath}^{\text{path-pos}}_{\text{reg}}$ query containment. Since the former is in EXPTIME (see [24], Theorem 8.4.), we obtain the desired result.

The lower bound follows from adapting known EXPTIME-complete results regarding the satisfiability of PDL versions close to XPath (see e.g. Section 4.4 of Alechina, N., Demri, S., & De Rijke, M. (2003). A modal perspective on path constraints. Journal of Logic and Computation, 13(6), 939-956.; or Theorem 8.4 in [24]). These results present reductions from the acceptance problem of a Turing Machine that decides a language in EXPTIME. The only difficulty in the adaptation of these proofs is dealing with a bounded alphabet, since the natural adaptation of these results would result in a reduction needing an unbounded alphabet. But this can be done by coding the symbols of the alphabet as binary strings– of unbounded length but now using a bounded alphabet–, as is repeatedly done in [4] (see the EXPSPACE-hardness proof). For example, if Σ contains 4 characters, then we treat them as strings 00, 01, 10 and 11.

□