Containment of Pattern-Based Queries over Data Trees

Claire David
Université Paris-Est Marne-la-Vallée
claire.david@univ-mlv.fr

Amélie Gheerbrant
University of Edinburgh
agheerbr@inf.ed.ac.uk

Leonid Libkin
University of Edinburgh
libkin@inf.ed.ac.uk

Wim Martens
University of Bayreuth
wim.martens@uni-bayreuth.de

ABSTRACT

We study static analysis, in particular the containment problem, for analogs of conjunctive queries over XML documents. The problem has been studied for queries based on arbitrary patterns, not necessarily following the tree structure of documents. However, many applications force the syntactic shape of queries to be tree-like, as they are based on proper tree patterns. This renders previous results, crucially based on having non-tree-like features, inapplicable. Thus, we investigate static analysis of queries based on proper tree patterns. We go beyond simple navigational conjunctive queries in two ways: we look at unions and Boolean combinations of such queries as well and, crucially, all our queries handle data stored in documents, i.e., we deal with containment over data trees.

We start by giving a general Π_2^p upper bound on the containment of conjunctive queries and Boolean combinations for patterns that involve all types of navigation through documents. We then show matching hardness for conjunctive queries with all navigation, or their Boolean combinations with the simplest form of navigation. After that we look at cases when containment can be witnessed by homomorphisms of analogs of tableaux. These include conjunctive queries and their unions over child and next-sibling axes; however, we show that not all cases of containment can be witnessed by homomorphisms. We look at extending tree patterns used in queries in three possible ways: with wildcard, with schema information, and with data value comparisons. The first one is relatively harmless, the second one tends to increase complexity by an exponential, and the last one quickly leads to undecidability.

EDBT/ICDT '13, March 18 - 22 2013, Genoa, Italy.

Categories and Subject Descriptors

H.2.3 [**Database management**]: Languages—*Query Languages*; F.2 [**Analysis of algorithms and problem complexity**]: General

General Terms

Algorithms, Theory

1. INTRODUCTION

Static analysis of queries and specifications has been actively investigated in the context of XML, not only due to its importance in tasks such as query optimization but also due to a very different nature of the results brought to the fore by the hierarchical structure of XML documents [1, 3, 7, 10, 13, 14, 16, 17, 18, 19, 28, 30, 33, 35]. Typical reasoning problems include consistency of queries or constraints with respect to schema information, typechecking of transformations, security of views, and crucially, query containment, with or without schema information. The latter is the problem we deal with. Starting from the relational case, we know that query containment is often the technical core of many query optimization [15]. In recent years query containment found multiple applications not only in query answering and optimization, but also in data integration, exchange, and provenance among others [22, 23, 27].

Already in the relational case, we know that, by and large, containment is decidable for conjunctive queries and relatives, and undecidable for expressive queries, such as those coming from the full relational algebra. In the XML case, static analysis of queries has been largely restricted to queries in various fragments of XPath [35] and analogs of conjunctive queries [10, 11, 21], primarily describing the structure of documents. The latter classified the complexity of query containment depending on the list of used axes, providing a seemingly complete picture.

Nonetheless, the picture depicted by [10, 11, 21] is not as complete as it seems. Firstly, this work basically took *relational conjunctive queries* on top of XML documents, and considered containment problems for them. Queries like that in a way do not follow tree patterns. An example of such

a query is one saying that we have two nodes s and s', so that s' is a descendant of s, and we have other nodes $s_1, \ldots, s_n$ so that each s_i is a descendant of s and an ancestor of s'. This says that the s_is appear in some order on the unique path from s and s'. But note that the query itself is DAG-shaped rather than tree-shaped (i.e., if we consider its tableau, it is a DAG rather than a tree). Secondly, results in those papers mainly concentrated on navigational features and much less on the data that documents carry. For instance, containment of queries with data values remained practically unexplored (as these papers concentrated on satisfiability).

And yet many applications demand proper *XML conjunctive queries* which are tree-based and can return data. Such queries are naturally induced by *XML patterns* which appear in multiple applications including XML data exchange, data integration, and query optimization [4, 5, 6, 8, 9, 24]. Such patterns are given by grammars or formation rules that naturally induce a tree structure. As a simple example, we can say that a label a is a pattern, and if $\pi_1, \ldots, \pi_n$ are patterns, then $a[\pi_1, \ldots, \pi_n]$ is a pattern. It will be matched by an a-labeled node that has n (not necessarily distinct) children matching $\pi_1, \ldots, \pi_n$. Such patterns (and more complex ones including other axes) form the basis for describing XML schema mappings and incompleteness in XML documents. And yet they cannot generate the DAG-like "confluence" behavior explained earlier; because for any two nodes appearing on a descendant path, it is always specified which one appears first. But it is precisely this behavior that is behind many of the complexity results applied to graph-based pattern queries in XML.

So our question is: what are the costs of static analysis problems for proper *tree-based pattern queries*?

A key feature of the main applications of such patterns is that they are not purely about the *structure* of documents, but they also collect *data*. For instance, a pattern $a(x)[b(x), c(y)]$ collects values x and y of data so that a document has an a-node with value x, that in turn has a b-child with the same value x and a c-child with value y. Thus, queries return sets of tuples, rather just a yes/no answer for the existence of a purely structural match.

Furthermore, we do not look solely at analogs of conjunctive queries. In the relational case, it is well known that containment of both conjunctive queries and their unions has the same complexity, namely NP-complete [15, 34], and the complexity of containment of arbitrary Boolean combinations of conjunctive queries moves one level up in the polynomial hierarchy to Π_2^p-complete. So we deal with analogs of such queries too, built from tree-based patterns.

Thus, our revised goal is to investigate the containment problems for analogs of conjunctive queries and relatives (unions, Boolean combinations) based on tree-based XML patterns over both structural information and data that XML documents carry.

Overview of the results. As an abstraction of XML documents with data values we use, as is common, *data trees*. In those trees, each node carries both a label and a data value. This is also sufficient to model XML documents whose nodes may have multiple attributes, simply by creating an extra child of a node for each attribute name.

We start by defining tree patterns upon which our queries are based. The simplest are patterns based on child navigation. An example is the pattern $a(x)[b(x), c(y)]$ that we explained before. Note that variables corresponding to data values are free: we view this pattern as $\pi(x, y)$, returning all pairs (x, y) such that a match occurs with x and y being the data values witnessing it. We can add horizontal navigation too, for instance we can have a pattern $a(x)[b(x) \rightarrow c(y)]$, stating that the c-witness is the sibling that follows the b-witness. More generally, we can have transitive closure axes too: for instance $a(x)[c(x)]//[b(x) \rightarrow^* c(y)]$ says that the a-node has a c-child with the same data value and two descendants, with labels b and c and data values x and y, so that they are siblings and the b-node occurs earlier in the sibling order. These types of patterns occur, for instance, in integration and exchange tasks for defining XML schema mappings [4, 6, 8] or in descriptions of XML with incompleteness features [2, 9].

Based on such patterns, we define conjunctive queries (CQs) by closing them under conjunction and existential quantification, i.e., as queries $q(\bar{x}) = \exists \bar{y} \bigwedge_i \pi_i(\bar{x}, \bar{y})$. We look at unions of conjunctive queries, or UCQs, which are of the form $\bigcup_i q_i(\bar{x})$, where each $q_i(\bar{x})$ is a CQ, and their Boolean combinations, or BCCQs, obtained by applying operations $q \cap q'$, $q \cup q'$, and $q - q'$ to CQs.

We present a general upper bound showing that containment of BCCQs that use all the axes is in Π_2^p. We show two matching lower bounds: either for BCCQs with the simplest navigation (only child relation), or CQs with all the axes.

In the relational case, CQ containment is tested by tableaux homomorphisms. The Π_2^p-hardness for general CQs precludes the possibility of such a test in general, but we show that with restricted sets of axes it is still possible (in fact we just have to exclude the horizontal $\rightarrow^*$ relation).

We then look at adding features to patterns and queries. First, we add wildcard and show that the Π_2^p-upper bound continues to hold. We also show that, for some classes of queries, the complexity of containment can jump from NP-complete to Π_2^p-complete if wildcard is added to patterns. Furthermore, we identify a 'safe' case of using wildcard, namely everywhere except at roots of patterns, that preserves homomorphism characterizations of containment.

The next addition we consider is containment under schema information (abstracted as a tree automaton, which capture many schema formalisms for XML). Here we show that the upper bound increases to double-exponential, and a matching lower bound can be shown for CQs with all axes. Finally, we look at adding data-value comparisons to queries, in particular the disequality ($\neq$) comparisons. This addition has a much more dramatic effect on the complexity of containment: it becomes undecidable for BCCQs, and for CQs when both comparisons and schemas are present (even with severe restrictions on available navigation).

Comparison with non-tree pattern queries. As we already mentioned, a number of results exist on CQs based on graph-shaped, rather than tree-shaped patterns [10, 11, 21]. None of those results extend to handle unions and differences of queries (i.e., UCQs and BCCQs) and they handle data values in a very limited way. Below we contrast them with our results.

For CQs that may contain arbitrary graph patterns, the Π_2^p upper bound continues to hold, but hardness requires less: for instance, purely navigational queries with non-tree patterns are already Π_2^p-complete for vertical navigation [11] (for tree patterns they stay in NP, as we show). Under schema, containment of CQs jumps to doubly-exponential too [10]. With $\neq$ comparisons, even CQ containment becomes undecidable [10].

Those results cover only a part of the landscape that we study here (i.e., CQs, concentrating mainly on pure navigation), and, crucially, under different assumptions on the shape of queries. Such assumptions make most existing proofs inapplicable for us (as they often rely heavily on non-tree features, such as the confluence, explained earlier, and the bidirectionality of axes).

Organization. We give key definitions in Section 2. The Π_2^p upper bound for BCCQs is shown in Section 3. In Section 4 we prove two matching lower bounds. In Section 5 we investigate cases when containment can be witnessed by the homomorphism of tableaux. Section 6 studies the effect of adding wildcard to queries. In Section 7 we investigate static analysis under schema constraints. Adding data-value comparison is studied in Section 8. Concluding remarks are given in Section 9. Due to space limitations, proofs are only sketched.

2. TREES, PATTERNS, AND QUERIES

Unranked trees and data trees. We start with the standard definitions of unranked finite trees which serve as an abstraction of XML documents when one deals with their structural properties. A finite unranked tree domain is a non-empty, prefix-closed finite subset D of $\mathbb{N}^*$ (words over $\mathbb{N}$) such that $s\cdot i \in D$ implies $s\cdot j \in D$ for all $j < i$ and $s \in \mathbb{N}^*$. We refer to elements of finite unranked tree domains as *nodes*. We assume a countably infinite set $\mathcal{L}$ of possible labels that can be used to label tree nodes. An unranked tree is a structure $\langle D, \downarrow, \rightarrow, \lambda\rangle$, where

- D is a finite unranked tree domain,
- $\downarrow$ is the child relation: $s \downarrow s\cdot i$ for $s\cdot i \in D$,
- $\rightarrow$ is the next-sibling relation: $s\cdot i \rightarrow s\cdot(i+1)$ for $s\cdot(i+1) \in D$, and
- $\lambda : D \rightarrow \mathcal{L}$ is the labeling function assigning a label to each node.

We denote the reflexive-transitive closure of $\downarrow$ by $\downarrow^*$ (descendant-or-self), and the reflexive-transitive closure of $\rightarrow$ by $\rightarrow^*$ (following-sibling-or-self).

Data trees are a standard abstraction of XML documents when one deals with both structural properties and data. Suppose we have a domain $\mathcal{D}$ of *data values*, such as strings, numbers, etc. A *data tree* is a structure $t = \langle D, \downarrow, \rightarrow, \lambda, \rho\rangle$, where $\langle D, \downarrow, \rightarrow, \lambda\rangle$ is an unranked tree, and $\rho : D \rightarrow \mathcal{D}$ assigns each node a data value. In XML documents, nodes may have multiple attributes, but this is easily modeled with data trees. For instance, to model a node with attributes $a_1, \ldots, a_n$ having values $v_1, \ldots, v_n$, we pick special labels $\ell_1, \ldots, \ell_n$, and create n extra children labeled $\ell_1, \ldots, \ell_n$ carrying values $v_1, \ldots, v_n$.

Patterns. As already explained, the patterns that we use are naturally tree-shaped. To explain how they are introduced, let us consider the reducts of data trees to the child relation, i.e., structures $\langle D, \downarrow, \lambda, \rho\rangle$. Trees of this form can be defined by recursion. That is, a node labeled with $a \in \mathcal{L}$ and carrying a data value $v \in \mathcal{D}$ is a data tree, and if $t_1, \ldots, t_n$ are trees, we can form a new tree by making them children of a node with label a and data value v.

In patterns we use also variables; the intention for them is to match data values in data trees. Thus, they are essentially partial tree descriptions with variables appearing in place of some data values. We assume a countable infinite set $\mathcal{V}$ of variables, disjoint from the domain of values $\mathcal{D}$. So the previous inductive definition gives rise to the definition of the simplest patterns we consider here:

$$\pi := a(x)[\pi, \ldots, \pi] \tag{1}$$

with $a \in \mathcal{L}$ and $x \in \mathcal{V} \cup \mathcal{D}$. Here the sequence in $[\ldots]$ could be empty. In other words, if $\pi_1, \ldots, \pi_n$ is a sequence of patterns (perhaps empty), $a \in \mathcal{L}$ and $x \in \mathcal{V} \cup \mathcal{D}$, then $a(x)[\pi_1, \ldots, \pi_n]$ is a pattern. If $\bar{x}$ is the list of all the variables used in a pattern π, we write $\pi(\bar{x})$.

We denote patterns from this class by $\mathbf{\Pi}(\downarrow)$. The semantics of $\pi(\bar{x})$ is defined with respect to a data tree $t = \langle D, \downarrow, \rightarrow, \lambda, \rho\rangle$, a node $s \in D$, and a valuation $\nu : \bar{x} \rightarrow \mathcal{D}$ as follows: $(t, s, \nu) \models a(x)[\pi_1(\bar{x}_1), \ldots, \pi_n(\bar{x}_n)]$ iff

- $\lambda(s) = a$ (the label of s is a);
- $\rho(s) = \begin{cases} \nu(x) & \text{if } x \text{ is a variable} \\ x & \text{if } x \text{ is a data value;} \end{cases}$
- there exist not necessarily distinct children $s\cdot i_1, \ldots, s\cdot i_n$ of s so that $(t, s\cdot i_j, \nu) \models \pi_j(\bar{x}_j)$ for each $j \leq n$ (recall that n could be 0, in which case this last item is not needed).

We write $(t, \nu) \models \pi(\bar{x})$ if there is a node s so that $(t, s, \nu) \models \pi(\bar{x})$ (i.e., a pattern is matched somewhere in the tree). Also if $\bar{v} = \nu(\bar{x})$, we write $t \models \pi(\bar{v})$ instead of $(t, \nu) \models \pi(\bar{x})$.

A natural extension for these simple patterns is to include both vertical and horizontal navigation. Again the intuition comes from defining data trees as follows: a node labeled with $a \in \mathcal{L}$ and carrying a data value $v \in \mathcal{D}$ is a data tree, and if $t_1, \ldots, t_n$ are trees, we can form a new tree by making them children of a node with label a and data value v, so that their roots are connected in the order $t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n$.

This leads to the definition of patterns in the class $\mathbf{\Pi}(\downarrow, \rightarrow)$:

$$\pi := a(x)[\pi \rightarrow \ldots \rightarrow \pi] \tag{2}$$

with $a \in \mathcal{L}$ and $x \in \mathcal{V} \cup \mathcal{D}$. Again the sequence in $[\ldots]$ could be empty. In other words, if $\pi_1, \ldots, \pi_n$ is a sequence of patterns (perhaps empty), $a \in \mathcal{L}$ and $x \in \mathcal{V} \cup \mathcal{D}$, then $a(x)[\pi_1 \rightarrow \pi_2 \rightarrow \ldots \rightarrow \pi_n]$ is a pattern. The last clause in the definition of the semantics of $\mathbf{\Pi}(\downarrow)$ is modified as follows:

- there exists a child $s \cdot i$ of s so that $(t, s \cdot i, \nu) \models \pi_1(\bar{x}_j)$, $(t, s \cdot (i+1), \nu) \models \pi_2(\bar{x}_2)$, ..., $(t, s \cdot (i+n-1), \nu) \models \pi_n(\bar{x}_n)$. In other words, it is consecutive children that witness the satisfaction of subpatterns.

Patterns in $\mathbf{\Pi}(\downarrow)$ and $\mathbf{\Pi}(\downarrow, \rightarrow)$ completely specify the structure of a tree (depending on the available axes) and, in particular, only express local properties of trees. We therefore also consider their more expressive versions with transitive closure axes $\downarrow^*$ (descendant) and $\rightarrow^*$ (following sibling). More precisely, following [4, 20], we define general patterns by the rules:

$$\begin{array}{lcl} \pi & := & a(x)[\mu, \ldots, \mu]//[\mu, \ldots, \mu] \\ \mu & := & \pi \rightsquigarrow \ldots \rightsquigarrow \pi \end{array} \tag{3}$$

Here a, x and π are as before, and μ stands for a *sequence*, i.e., a forest such that the roots of its trees are sequential siblings in a tree, and each $\rightsquigarrow$ is either $\rightarrow$ or $\rightarrow^*$.

The class of such patterns is denoted by $\mathbf{\Pi}(\Downarrow, \Rightarrow)$, with $\Downarrow$ we use both types of downward navigation ($\downarrow$ and $\downarrow^*$) and $\Rightarrow$ meaning that we use both types of horizontal navigation ($\rightarrow$ and $\rightarrow^*$). The semantics is extended as follows.

- $(t, s, \nu) \models \pi_1 \rightsquigarrow \ldots \rightsquigarrow \pi_m$ if there is a sequence $s = s_1, \ldots, s_m$ of nodes so that $(t, s_i, \nu) \models \pi_i$ for each $i \leq m$ and $s_i \rightarrow s_{i+1}$ whenever the ith $\rightsquigarrow$ is $\rightarrow$, and $s_i \rightarrow^* s_{i+1}$ whenever the ith $\rightsquigarrow$ is $\rightarrow^*$.
- $(t, s, \nu) \models a(x)[\mu_1, \ldots, \mu_n]//[\mu'_1, \ldots, \mu'_k]$ if the satisfaction of $a(x)$ in node s is as before, and there exist n not necessarily distinct children $s_1, \ldots, s_n$ of s such that $(t, s_i, \nu) \models \mu_i$ for each $i \leq n$, and there exist k not necessarily distinct descendants $s'_1, \ldots, s'_k$ of s such that $(t, s'_i, \nu) \models \mu'_i$ for each $i \leq k$.

Notice that the semantics of patterns allows different μ_i to be mapped into the same nodes in a tree.

Finally, we consider a class $\Pi(\Downarrow)$ of patterns which is a restriction of the most general patterns to downward navigation only. These are defined by the grammar

$$\pi := a(x)[\pi, \ldots, \pi]//[\pi, \ldots, \pi] \tag{4}$$

where each of the sequences of patterns can be empty. That is, a pattern $a(x)[\pi_1, \ldots, \pi_n]//[\pi'_1, \ldots, \pi'_k]$ is witnessed in an a-labeled node assigning its data value to x if it has n children (not necessarily distinct) witnessing $\pi_1, \ldots, \pi_n$ and k descendants (again not necessarily distinct) witnessing $\pi'_1, \ldots, \pi'_k$.

Shorthands. We shall be using standard shorthand notations: $a(x)/\pi$ stands for $a(x)[\pi]$, while $a(x)//\pi$ denotes $a(x)//[\pi]$, and $a(x)/\pi//\pi'$ stands for $a(x)[\pi]//[\pi']$.

Conjunctive queries, their unions, and Boolean combinations. Pattern-based conjunctive XML queries are obtained by closing patterns by conjunction and existential quantification. Since we have different classes of patterns $\mathbf{\Pi}(\sigma)$, for σ being $\downarrow$, or $\downarrow, \rightarrow$, or $\Downarrow$, or $\Downarrow, \Rightarrow$, we have different classes of conjunctive queries denoted by $\mathrm{CQ}(\sigma)$. More precisely, $\mathrm{CQ}(\sigma)$ queries are of the form:

$$q(\bar{x}) \;=\; \exists \bar{y} \bigwedge_{i=1}^{n} \pi_i(\bar{z}_i) \tag{5}$$

where each π_i is a $\mathbf{\Pi}(\sigma)$ pattern, and each $\bar{z}_i$ is contained in $\bar{x}, \bar{y}$. The semantics is standard: $(t, \nu) \models q(\bar{x})$ if there is an extension ν' of valuation ν to variables $\bar{y}$ such that $(t, \nu') \models \pi_i(\bar{z}_i)$ for every $i \leq n$.

As is standard, we also write $t \models q(\bar{v})$ if $(t, \nu) \models q(\bar{x})$ with $\nu(\bar{x}) = \bar{v}$.

Of course conjunctive queries are closed under conjunction. Standard ways of enriching their power include considering unions of conjunctive queries (or UCQs, which, in the relational case, capture the positive fragment of relational algebra) and more generally, Boolean combinations of conjunctive queries (or BCCQs, which, while possessing some form of negation, still retain many nice properties that relational algebra as a whole loses).

Formally, a query from $\mathrm{UCQ}(\sigma)$ is of the form $q(\bar{x}) = q_1(\bar{x}) \cup \ldots \cup q_m(\bar{x})$, where each $q_i(\bar{x})$ is a $\mathrm{CQ}(\sigma)$ query. It returns the union of answers to the q_i's, i.e., $(t, \nu) \models q(\bar{x})$ iff $(t, \nu) \models q_i(\bar{x})$ for some $i \leq m$.

Queries in the class BCCQ are obtained as follows: take some queries $q_1(\bar{x}), \ldots, q_m(\bar{x})$ from $\mathrm{CQ}(\sigma)$ and consider a Boolean combination of them, i.e., close them under operations $q \cap q'$, $q \cup q'$, and $q - q'$. The semantics is extended naturally, with those interpreted as intersection, union, and set difference, respectively.

The answer to a query $q(\bar{x})$, from any of the above classes, on a data tree t is defined as $q(t) = \{\nu(\bar{x}) \mid (t, \nu) \models q(\bar{x})\}$. Note that our definitions of query classes ensure that $q(t)$ is always finite.

Containment. The main problem we study here is the containment problem. Given two queries $q(\bar{x}), q'(\bar{x}')$ with tuples of free variables of the same length, we write $q \subseteq q'$ iff $q(t) \subseteq q'(t)$ for every data tree t. So the problem we look at is the following.

PROBLEM:	$\mathrm{CQ}_{\subseteq}(\sigma)$
INPUT:	queries $q(\bar{x}), q'(\bar{x}')$ in $\mathrm{CQ}(\sigma)$;
QUESTION:	is $q \subseteq q'$?

If instead of queries in $\mathrm{CQ}(\sigma)$ we use queries in $\mathrm{UCQ}(\sigma)$, we refer to the problem $\mathrm{UCQ}_{\subseteq}(\sigma)$ and, if we use queries from $\mathrm{BCCQ}(\sigma)$, we refer to the problem $\mathrm{BCCQ}_{\subseteq}(\sigma)$.

In the *relational* case, these problems are among the basic problems of database theory. The complexity of $\mathrm{CQ}_{\subseteq}$ and $\mathrm{UCQ}_{\subseteq}$ over relational databases is NP-complete [15, 34] (under the representation of UCQs that we use here), and the complexity of $\mathrm{BCCQ}_{\subseteq}$ is Π_2^p-complete [34].

3. AN UPPER BOUND

A priori, there is no upper bound that is immediate for the containment problem. In fact, in the presence of negation (even a limited form of it) combined with XML hierarchical structure, some reasoning problems can become undecidable (see, e.g., [17, 4]). In the relational case, we know that containment for BCCQs is Π_2^p-complete, but this does not imply the same bounds for XML pattern-based queries, especially those that might use transitive closure axes $\rightarrow^*$ and $\downarrow^*$.

Nevertheless, we can show that for all such queries, the containment problem remains not only decidable, but the upper bound on its complexity continues to match that for the simplest relational queries. In fact we show the following.

THEOREM 3.1. *The problem* $\mathrm{BCCQ}_{\subseteq}(\Downarrow, \Rightarrow)$ *is decidable in* Π_2^p.

In other words, for each of the classes of queries — CQ, UCQ, BCCQ— and for each of the classes of patterns seen so far, the containment problem is in Π_2^p, as all of these problems are subsumed by the containment problem of BCCQs with $\mathbf{\Pi}(\Downarrow, \Rightarrow)$-patterns.

Proof sketch. Checking whether $q_1 \subseteq q_2$ is the same as checking $q_1 - q_2 = \emptyset$, so it will suffice to give a Σ_2^p algorithm for checking if a BCCQ q returns a nonempty result on some data tree. We assume that q is a Boolean combination of CQs $q_1, \ldots, q_m$. For the sketch we assume they are Boolean (free variables do not change anything). To check satisfiability it suffices to guess an assignment $\chi : \{1, \ldots, m\} \rightarrow \{0, 1\}$ so that for

$$q' = \bigwedge\{q_i \mid \chi(i) = 1\} \text{ and } q'' = \bigvee\{q_j \mid \chi(j) = 0\}$$

we have a tree t such that $q'(t)$ is true and $q''(t)$ is false. Note that q' is a CQ, and q'' is a UCQ. The idea of the proof is to turn this into a certain answer problem in XML data exchange [8]. We let schemas of XML documents be arbitrary and the mapping consist of a single rule $\_ \rightarrow q'$, forcing the patterns of q' in every target tree. Then we check whether the certain answer to q'' is false: this happens iff there is a tree satisfying q' and the negation of q''.

The latter requires two steps in the proof. One is a modification of the proof of the coNP *data* complexity of certain answers in [8]. The problem is that the latter proof produces a witnessing tree whose size is exponential in q'', which is too large for our purposes. So we show how to encode the exponential witness by a data structure whose size is polynomial in q', q'' and which allows checking for satisfiability of UCQs. The second step is making sure that all the guesses are combined in the right order to yield a Σ_2^p algorithm. □

4. LOWER BOUNDS FOR CONTAINMENT

Now that we know that all the containment problems are in Π_2^p, it is natural to ask when we have matching lower bounds. Note that in all the variations of containment problems, we have two parameters: the class of queries (going from the simplest, CQs, to UCQs, and to BCCQs), and the set of axes (again, starting with the simplest, just $\downarrow$, and then going to more complex $\downarrow, \rightarrow$, as well as $\Downarrow$ and $\Downarrow, \Rightarrow$).

What we show in this section is that each of the combination simplest/hardest leads to Π_2^p-hardness. That is, the containment problem with the simplest of axes, just $\downarrow$, is Π_2^p-complete if we allow Boolean combinations of queries. If we have just CQs, the containment becomes Π_2^p-complete when we have all the axes, i.e. $\downarrow, \downarrow^*, \rightarrow$, and $\rightarrow^*$.

Note that the first result on the surface is rather similar to Π_2^p-completeness of containment of relational BCCQs [34]. Indeed, the standard representation of relations in XML only needs the $\downarrow$ axis, and shallow documents. However, the result does not follow from the results in [34], as we demand containment over all XML documents, not only those that properly represent relational databases of a given schema. In particular, if we have two relational BCCQs q and q', and their natural XML codings as $\mathrm{BCCQ}(\downarrow)$ queries q_{XML} and q'_{XML}, then $q_{\mathrm{XML}} \subseteq q'_{\mathrm{XML}}$ implies $q \subseteq q'$ (as each relational database can be coded as an XML tree), but under the same coding $q \subseteq q'$ need not imply $q_{\mathrm{XML}} \subseteq q'_{\mathrm{XML}}$.

Even though we cannot use results on [34], we can modify reductions to apply to all XML documents and obtain the following.

THEOREM 4.1. *The problem* $\mathrm{BCCQ}_{\subseteq}(\downarrow)$ *is* Π_2^p*-complete.*

Next, we move to the other extreme case: CQs with all the axes. Of course relational containment of CQs is NP-complete, so to get hardness for a larger class, one has to use, in an essential way, the hierarchical structure of XML. In fact we provide a rather elaborate reduction showing that the navigational abilities of all the axes are sufficient to increase the complexity even of conjunctive query containment.

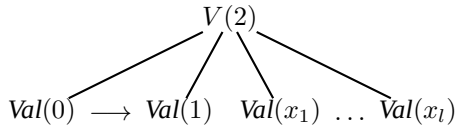
THEOREM 4.2. *The problem* $CQ_{\subseteq}(\Downarrow, \Rightarrow)$ *is* Π_2^p*-complete.*

Proof sketch. The upper bound was shown in the previous section. To show hardness, we proceed by reduction from ∀∃3CNF. Given such a formula $\varphi := \forall p_1 \ldots \forall p_l \exists r_1 \ldots \exists r_m \bigwedge_i (\ell_{i1} \vee \ell_{i2} \vee \ell_{i3})$, where the ℓ_{ij}s could be positive or negative literals, we associate with it two Boolean queries $q, q' \in \mathrm{CQ}(\Downarrow, \Rightarrow)$ such that φ is true if and only if $q \subseteq q'$.

We construct q and q' so that for every possible valuation v of the p_is, two conditions hold. First, there exists a tree t_v satisfying q which encodes v. Second, such a tree t_v satisfies q' iff there is a valuation v^+ extending v to the r_is and for

which φ evaluates to true. The key idea behind the construction is encoding possible valuations for quantified variables, and we explain it now. The encoding of the CNF formula itself is standard.

In order to encode every possible valuation of the p_is using one single query q, we associate a variable x_i to each p_i and then take full advantage of navigational features to model assignments. Specifically, we use a tree pattern $V(2)/[\mathit{Val}(0) \to \mathit{Val}(1), \mathit{Val}(x_1), \ldots, \mathit{Val}(x_l)]$. Its root has $l+2$ children, among which the ordering is specified for two ($\mathit{Val}(0) \to \mathit{Val}(1)$). The remaining l children carry the x_is, but note that their exact positions as children of the $V(2)$ node are not specified. This is illustrated below:



Now on every complete tree t witnessing this pattern via some homomorphism h, the image of every x_i will either be on the left, or on the right of 0, i.e., either

$$t \models V(2)[\mathit{Val}(h(x_i)) \to^* \mathit{Val}(0)],$$

or

$$t \models V(2)[\mathit{Val}(0) \to^* \mathit{Val}(h(x_i))].$$

This allows us to associate a valuation v of the p_is to any tree satisfying this pattern by letting $v(p_i)$ be false if the image of x_i occurs on the left of $\mathit{Val}(0)$, and by letting $v(p_i)$ be true otherwise. The rest of the encoding consists of the standard encoding of a CNF formula, and ensuring, for q', that the extended valuation makes that formula true. □

Remark Note that letting one omit a complete specification of the sibling ordering has the effect of encoding 2^n possible valuations with n different nodes. This is similar to the effect of using "confluence" features in [11]. In both cases, such a concise encoding of exponentially many valuations led to Π_2^p lower bounds.

5. CONTAINMENT VIA HOMOMORPHISMS

A classical result of relational database theory says that containment of relational CQs is NP-complete and containment is witnessed by the existence of a *homomorphism* of tableaux: if T_i is the tableau of a query q_i, for $i = 1, 2$, then $q_1 \subseteq q_2$ iff there is a homomorphism from T_2 to T_1 [15]. However, the results of the previous section indicate that such a characterization of containment via homomorphisms cannot be extended to all classes of CQs we consider here. Indeed, testing for the existence of a homomorphism is a classical NP-complete problem and we saw in Theorem 4.2 that containment of CQ($\Downarrow, \Rightarrow$) queries is Π_2^p-complete.

So the question is: for what types of queries, if any, can we characterize containments via homomorphisms of their tableaux? And even before answering this question, we need to ask: what are the tableaux of XML-based CQs?

Since tableaux for relational queries are essentially incomplete databases (more precisely, naïve tables with a distinguished row of variables), it is natural to define tableaux of XML CQs as incomplete XML trees. Indeed, patterns forming a query are essentially incompletely specified trees, so we can view each query as an incomplete tree (more precisely, a forest). The theory of incompleteness of XML has been developed [2, 9] and thus we can borrow a notion of an incomplete tree.

Incomplete trees and homomorphism. An *incomplete tree* is defined as a structure $t = (N, V, \downarrow, \downarrow^*, \rightarrow, \rightarrow^*, \lambda, \rho)$, where

- N and V are disjoint finite sets of the nodes of t and its data values, respectively; we assume that $V \subset \mathcal{D} \cup \mathcal{V}$, i.e., values could be either data values or variables;
- all of $\downarrow, \downarrow^*, \rightarrow, \rightarrow^*$ are binary relations on N;
- λ is a partial function from N to $\mathcal{L}$; and
- ρ is a function from N to V.

Note that in an incomplete tree, the relations $\downarrow, \downarrow^*, \rightarrow, \rightarrow^*$ may be interpreted arbitrarily. In particular, some incomplete trees cannot be extended to a complete tree. The issue is discussed in details in [9]. The labeling function is partial, reflecting the fact that labels of some nodes may not be known. The data assigning function ρ is not partial since some data values could be variables, just like in patterns.

Given two incomplete trees $t = \langle N, V, \downarrow, \downarrow^*, \rightarrow, \rightarrow^*, \lambda, \rho \rangle$ and $t' = \langle N', V', \downarrow, \downarrow^*, \rightarrow, \rightarrow^*, \lambda', \rho' \rangle$, a *homomorphism* from t to t' is a map $h : N \cup V \to N' \cup V'$ such that:

- $h(N) \subseteq N'$ and $h(V) \subseteq V'$;
- if wRw' in t, with $w, w' \in N$ and R one of the relations $\downarrow, \downarrow^*, \rightarrow, \rightarrow^*$, then $h(w)Rh(w')$ in t';
- if $\lambda(w)$ is defined in t, then $\lambda'(h(w)) = \lambda(w)$;
- h is the identity on elements of $\mathcal{D}$; and
- $h(\rho(w)) = \rho'(h(w))$ for all $w \in N$.

Note that each tree can be viewed as an incomplete tree (with the natural interpretations of the binary relations) and thus it makes sense to speak of a homomorphism from an incomplete tree to a complete tree.

Our plan is now as follows. We show how to associate, to a CQ q, an incomplete tree t_q. If q is a Boolean query, then $t \models q$ iff there is a homomorphism from t_q into t. If q has free variables $\bar{x}$, then $t \models q(\bar{v})$ iff there is a homomorphism from $t_{q(\bar{x})}$ to t that sends $\bar{x}$ to $\bar{v}$.

We then show that, for some classes σ of axes and queries $q, q' \in \mathrm{CQ}(\sigma)$, we have $q \subseteq q'$ iff there is a homomorphism from the σ-restriction of $t_{q'}$ to the σ-restriction of t_q.

Incomplete trees of CQs. We now define analogs of tableaux of relational CQs; these will be incomplete trees.

We first define an incomplete tree t_π for each pattern π. To carry the inductive construction, we shall need to define both trees t_π and t_μ for sequences μ. Note that even though we use the name 'incomplete tree', such a structure need not be a tree (due to incompleteness); in fact t_μs will be forest-like. Each incomplete tree t of the form t_π or t_μ will have a set $\text{RT}(t)$ of roots associated with it in such a way that $\text{RT}(t_\pi)$ is always a singleton. The inductive construction is as follows.

- If $\pi = a(x)$, then $t_\pi = \langle\{s\}, \{x\}, \downarrow, \downarrow^*, \rightarrow, \rightarrow^*, \lambda, \rho\rangle$, where s is a single node, all the binary relations are empty, $\lambda(s) = a$ and $\rho(s) = x$. Furthermore, $\text{RT}(t_\pi) = \{s\}$.
- Let $\pi = a(x)[\mu_1, \ldots, \mu_n]//[\mu'_1, \ldots, \mu'_k]$. Suppose we already have t_{μ_i}s and $t_{\mu'_j}$s defined. Let N_i and V_i be the sets of nodes and values in t_{μ_i}s and N'_j and V'_j be the sets of nodes and values in $t_{\mu'_j}$s. By renaming nodes in those incomplete trees, we may assume that all the sets N_is and N'_js are disjoint. Then
$$t_\pi = (N, V, \downarrow, \downarrow^*, \rightarrow, \rightarrow^*, \lambda, \rho)$$
where $N = \{s\} \cup \bigcup_i N_i \cup \bigcup_j N'_j$, with s being a new node, and $V = \bigcup_i V_i \cup \bigcup_j V'_j$. The binary relations are the unions of those relations in the t_{μ_i}s and $t_{\mu'_j}$s. In addition, we put:
  - $s \downarrow s'$ for each $s' \in \text{RT}(\mu_i)$, for $i \leq n$; and
  - $s \downarrow^* s'$ for each $s' \in \text{RT}(\mu'_j)$, for $j \leq k$.

  The functions λ and ρ are the same as in the t_{μ_i}s and $t_{\mu'_j}$s; in addition $\lambda(s) = a$ and $\rho(s) = x$. Furthermore, $\text{RT}(t_\pi) = \{s\}$.
- Let $\mu = \pi_1 \rightsquigarrow \ldots \rightsquigarrow \pi_n$. Let t_{π_i} be an incomplete tree $\langle N_i, V_i, \downarrow, \downarrow^*, \rightarrow, \rightarrow^*, \lambda_i, \rho_i\rangle$. As before, assume that by renaming nodes, all the N_is are disjoint. Let $\text{RT}(t_{\pi_i}) = \{s_i\}$.
  Then $t_\mu = \langle N, V, \downarrow, \downarrow^*, \rightarrow, \rightarrow^*, \lambda, \rho\rangle$, where $N = \bigcup_i N_i$ and $V = \bigcup_i V_i$; the binary relations are unions of those in the t_{π_i}s, and in addition we put:
  - $s_i \rightarrow s_{i+1}$ if μ contains $\pi_i \rightarrow \pi_{i+1}$; and
  - $s_i \rightarrow^* s_{i+1}$ if μ contains $\pi_i \rightarrow^* \pi_{i+1}$.

  The functions λ and ρ coincide with λ_i and ρ_i on N_i. Moreover, $\text{RT}(\mu) = \{s_1, \ldots, s_n\}$.

With a query
$$q(\bar{x}) = \exists \bar{y_1} \ldots \exists \bar{y_n} \pi_1(\bar{x}, \bar{y_1}) \wedge \ldots \wedge \pi_n(\bar{x}, \bar{y_n})$$
we associate an incomplete data tree
$$t_q = (N, V, \downarrow, \downarrow^*, \rightarrow, \rightarrow^*, \lambda, \rho)$$
which is the node-disjoint union of all the t_{π_i}s; that is, we rename nodes so that their sets are disjoint (but not the values), and take the union of structures $t_{\pi_1}, \ldots, t_{\pi_n}$.

The incomplete trees t_q indeed play the role of tableaux of CQs. Recall that in the relational case, we have $D \models q(\bar{v})$ iff there is a homomorphism from the tableau of $q(\bar{x})$ to D that sends $\bar{x}$ to $\bar{v}$. The same is true here. The result is very similar to one in [9], adapted to the definitions given here.

PROPOSITION 5.1. *Let t be a data tree, and $q(\bar{x})$ a query from CQ$(\Downarrow, \Rightarrow)$. Then $t \models q(\bar{v})$ iff there is a homomorphism $h : t_q \rightarrow t$ so that $h(\bar{x}) = \bar{v}$.*

Containment and homomorphisms. We already mentioned that a classical result of relational database theory states that relational CQ containment $q \subseteq q'$ holds iff the tableau of q' can be homomorphically mapped into the tableau of q. Furthermore, an analog of this cannot possibly hold for queries in CQ$(\Downarrow, \Rightarrow)$ unless some complexity classes collapse. Nonetheless, it will work for queries that do not use all the axes.

Suppose we have a query q from CQ$(\downarrow)$. Then its incomplete tree t_q records no information about $\downarrow^*$, $\rightarrow$, and $\rightarrow^*$. So for two such queries q and q', a homomorphism of the $\downarrow$-reducts of t_q and $t_{q'}$ (that only keep information about $\downarrow$, λ, and ρ) is the same as a homomorphism t_q and $t_{q'}$. Hence, even for queries that use reduced sets of axes, e.g., CQ$(\downarrow)$ or CQ$(\downarrow, \rightarrow)$, we can still meaningfully talk about homomorphisms of their incomplete trees, in place of homomorphisms of reducts of incomplete trees.

We next show that without transitive closure axes, we have an analog of relational containment.

THEOREM 5.2. *Let $q(\bar{x})$ and $q'(\bar{x}')$ be two queries from either CQ$(\downarrow)$, or CQ$(\downarrow, \rightarrow)$. Then $q \subseteq q'$ iff there is a homomorphism $h : t_{q'} \rightarrow t_q$ so that $h(\bar{x}') = \bar{x}$.*

Since testing homomorphism existence is done in NP, and NP-hardness bound for relational CQs trivially applies to CQ$(\downarrow)$ queries, we obtain the following.

COROLLARY 5.3. *The containment problems for CQ$(\downarrow)$ and CQ$(\downarrow, \rightarrow)$, i.e., CQ$_\subseteq(\downarrow)$ and CQ$_\subseteq(\downarrow, \rightarrow)$, are* NP*-complete.*

In fact, we prove an even more general result, that shows the applicability of the homomorphism technique to queries in CQ$(\Downarrow, \rightarrow)$, i.e., queries using all forms of vertical navigation, but only the next-sibling form of horizontal navigation. Formally, they are CQs based on patterns from $\mathbf{\Pi}(\Downarrow, \rightarrow)$ defined as
$$\begin{array}{lcl} \pi & := & a(x)[\mu, \ldots, \mu]//[\mu, \ldots, \mu] \\ \mu & := & \pi \rightarrow \ldots \rightarrow \pi \end{array} \quad (6)$$
That is, they extend $\mathbf{\Pi}(\downarrow, \rightarrow)$ patterns by allowing descendants, and prohibiting only $\rightarrow^*$.

Given a query $q \in$ CQ$(\Downarrow, \rightarrow)$, we define an incomplete tree $(t_q)^*$ by replacing the interpretation of $\downarrow^*$ in t_q by the reflexive-transitive closure of the union of $\downarrow$ and $\downarrow^*$ in t_q. Then containment can be tested by the existence of homomorphisms between such extended tableaux. As an example, consider queries $q \subseteq q'$, where $q = \exists x\, a(x)//b(x)[c(x)]$ and $q' = \exists x\, a(x)//c(x)$. While there is no homomorphism from

$t_{q'}$ to t_q, there is one from $(t_{q'})^*$ to $(t_q)^*$. Indeed, in both structures there is a descendant axis going from the a-labeled node to the c-labeled node.

THEOREM 5.4. *Let $q(\bar{x})$ and $q'(\bar{x}')$ be two queries from CQ$(\Downarrow, \rightarrow)$. Then $q \subseteq q'$ iff there is a homomorphism $h : (t_{q'})^* \rightarrow (t_q)^*$ so that $h(\bar{x}') = \bar{x}$.*

Proof sketch. The right to left direction of the equivalence is immediate. To show the other direction, we assume $q \subseteq q'$ and we turn the forest t_q into some "canonical" complete tree T such that there is a natural one to one homomorphism $h_1 : (t_q)^* \rightarrow T$ and such that for all $w, w' \in (t_q)^*$, for all $R \in \{\rightarrow, \downarrow, \downarrow^*\}$, we have wRw' iff $h_1(w)Rh_1(w')$. To this end, we create new nodes labeled with a fresh label $\heartsuit$ and a fresh data value $\sharp$. One of these nodes becomes the common parent of each of the roots of the tree patterns in t_q and thus becomes the root of T. We also define a recursive procedure replacing descendant axis $w_1 \downarrow^* w_2$ occurring in t_q by child paths $w_1 \downarrow w_3 \downarrow w_2$, where w_3 is one of the new nodes labeled $\heartsuit(\sharp)$. We proceed in a similar way with sequences of siblings which are given as mere unions. We order them arbitrarily using the next-sibling relation, but we always take care of inserting one of the new $\heartsuit(\sharp)$-labeled nodes in between two siblings which were not previously related by a $\rightarrow$-arrow. We finally substitute fresh distinct constants for every distinct variable, thus obtaining a complete tree. From $q \subseteq q'$, we then infer that there exists another homomorphism $h_2 : t_{q'} \rightarrow T$. Relying on the special properties of h_1, we finally construct the homomorphism $h : (t_{q'})^* \rightarrow (t_q)^*$ from h_1 and h_2 by letting $h(x) = h_1^{-1}(h_2(x))$. □

As before, we immediately obtain the following.

COROLLARY 5.5. *The problem CQ$_\subseteq(\Downarrow, \rightarrow)$ is* NP*-complete.*

As mentioned earlier, replacing $\rightarrow$ by $\Rightarrow$ and obtaining an analog of Theorem 5.4 is impossible without an unlikely collapse of complexity classes.

COROLLARY 5.6. *Assume that there is a polynomial-time algorithm that associates with each query $q \in$ CQ$(\Downarrow, \Rightarrow)$ an incomplete tree $t(q)$ so that $q \subseteq q'$ iff there is a homomorphism $t(q') \rightarrow t(q)$. Then* NP =coNP.

Indeed, since containment of CQ$(\Downarrow, \Rightarrow)$ is Π_2^p-hard and testing homomorphism existence is NP-complete, the existence of such a containment test would imply $\Pi_2^p \subseteq$ NP from which NP =coNP follows easily.

Polynomial-time cases. Our characterization of containment via homomorphisms immediately shows how to obtain polynomial-time cases of containment. Indeed, since containment is now reduced to the existence of homomorphisms, it is effectively cast as a constraint satisfaction (or conjunctive query evaluation) problem. Thus, we can use multiple known results classifying tractable cases of those and apply them to structures representing incomplete data trees. As all of these are quite routine, we leave the complete treatment to the full version (due to space limitations here), and now give just a couple of examples. One is the containment $q \subseteq q'$ for any of the classes CQ$(\downarrow)$, CQ$(\downarrow, \rightarrow)$, and CQ$(\Downarrow, \rightarrow)$ if the query q' is fixed. The other is containment for the classes CQ$(\downarrow)$ and CQ$(\downarrow, \rightarrow)$ when q' mentions each variable at most once (since in this case containment can be reduced to the combined complexity of evaluating conjunctive queries of fixed treewidth). More results will be provided in the full version.

Extension to unions of CQs. A classical result in relational theory says that for unions of relational conjunctive queries, $q = q_1 \cup \ldots \cup q_m$ and $q' = q'_1 \cup \ldots \cup q'_k$, we have $q \subseteq q'$ iff for every $i \leq m$, there exists $j \leq k$ so that $q_i \subseteq q'_j$ [34]. We call this the *SY-criterion* (for Sagiv/Yannakakis) for containment of UCQs. In particular, the SY-criterion implies that the complexity of containment of relational UCQs remains NP-complete (assuming, of course, that they are represented in the above way, as unions of CQs; for other syntactic representations, in which the union is not the outermost operation, the complexity is Π_2^p-complete [34]).

Note that we have defined XML queries in UCQ(σ) to be syntactically of the form $q_1 \cup \ldots \cup q_m$, where each q_i is a CQ(σ)-query. It turns out that for the classes which permit testing containment by means of homomorphisms between incomplete trees t_q, a similar extension to unions continues to be true.

PROPOSITION 5.7. *Queries in* UCQ$(\downarrow)$ *and* UCQ$(\downarrow, \rightarrow)$ *satisfy the SY-criterion for containment.*

This immediately gives us the following.

COROLLARY 5.8. *The problems* UCQ$_\subseteq(\downarrow)$ *and* UCQ$_\subseteq(\downarrow, \rightarrow)$ *are* NP*-complete.*

Indeed, for queries $q = q_1 \cup \ldots \cup q_m$ and $q' = q'_1 \cup \ldots \cup q'_k$ we simultaneously guess a map $f : \{1, \ldots, m\} \rightarrow \{1, \ldots, k\}$, and m maps h_i from $t_{q'_{f(i)}}$ to t_{q_i} for each $i \leq m$, and check, in polynomial time, if the h_is satisfy conditions of Theorem 5.2.

6. THE EFFECT OF WILDCARD

A standard feature of most XML formalisms is the use of *wildcard*, i.e., a special symbol in place of a label that matches every label in a tree. We normally use _ for wildcard. So patterns can be extended in the following way: instead of a pattern that starts with $a(x)$, we can have a pattern that starts with $\_(x)$. It will be witnessed in a node s of a data tree t even if we drop the requirement that labels match. When we deal with classes of patterns $\mathbf{\Pi}(\sigma)$ extended with wildcard, we write $\mathbf{\Pi}(\sigma, \_)$.

For instance, patterns in $\mathbf{\Pi}(\downarrow, \_)$ are given by

$$\pi := a(x)[\pi, \ldots, \pi], \quad a \in \mathcal{L} \cup \{\_\}, \quad x \in \mathcal{V} \cup \mathcal{D}. \qquad (7)$$

The semantics is extended, compared to (1), as follows. For a data tree $t = \langle D, \downarrow, \rightarrow, \lambda, \rho \rangle$, a node $s \in D$,

and a valuation $\nu : \bar{x} \to \mathcal{D}$, we have $(t, s, \nu) \models a(x)[\pi_1(\bar{x}_1), \ldots, \pi_n(\bar{x}_n)]$ iff

- $\lambda(s) = a$ if $a \in \mathcal{L}$;
- $\rho(s)$ is $\nu(x)$ if x is a variable, and x if x is a constant data value;
- there exist not necessarily distinct children $s \cdot i_1, \ldots, s \cdot i_n$ of s so that $(t, s \cdot i_j, \nu) \models \pi_j(\bar{x}_j)$ for each $j \leq n$.

Likewise we define all other classes of patterns extended with wildcard, e.g., $\mathbf{\Pi}(\downarrow, \to, \_)$ and $\mathbf{\Pi}(\Downarrow, \Rightarrow, \_)$, and classes of CQs, UCQs, and BCCQs based on them. For those queries we define the containment problem: for instance, $\mathrm{BCCQ}_{\subseteq}(\Downarrow, \Rightarrow, \_)$ is the problem of checking containment of BCCQs based on patterns from $\mathbf{\Pi}(\Downarrow, \Rightarrow, \_)$.

The question is then whether the use of wildcard increases the cost of testing containment. The first instance of that question is whether we can preserve the Π_2^p upper bound for all containment cases. The answer to this is positive. In fact, our proof of Theorem 3.1 already shows how to handle wildcard.

PROPOSITION 6.1. *The problem* $\mathrm{BCCQ}_{\subseteq}(\Downarrow, \Rightarrow, \_)$ *is in* Π_2^p.

Hence, all other containment problems are in Π_2^p in the presence of wildcard.

What does change, however, is the lower bounds. Recall that we saw in Corollary 5.8 that $\mathrm{UCQ}_{\subseteq}(\downarrow, \to)$ is in NP. The presence of wildcard makes the complexity jump: adding wildcards to $\mathbf{\Pi}(\downarrow, \to)$ patterns makes the complexity of containment of UCQs Π_2^p-hard, rather than being in NP.

THEOREM 6.2. *The problem* $\mathrm{UCQ}_{\subseteq}(\downarrow, \to, \_)$ *is* Π_2^p*-complete.*

Proof sketch. To show hardness, we adapt the lower bound proof of Theorem 4.2 by constructing queries $q_{\mathit{rigid}}, q'_{\mathit{rigid}}$ instead of q, q'. Recall that the query q was encoding all the possible valuations of the p_is using a special pattern over $\mathbf{\Pi}(\Downarrow, \Rightarrow)$. Additionally we used another pattern in q to encode the clauses in φ. We did not describe this pattern in the sketch of Theorem 4.2, but it is enough for the current sketch to note that it can alternatively be represented as a $\mathbf{\Pi}(\downarrow, \to)$-pattern π_φ. We define q_{rigid} by adding to π_φ two new nodes as first and second child of its root. These new nodes are respectively labeled *Val*(0) and *Val*(1). Let π_φ^{01} be the resulting pattern. For every $1 \leq i \leq l$, we also create a single node pattern labeled *Val*(x_i) and we form q_{rigid} by existentially quantifying the x_i's and taking the conjunction of these $l + 1$ patterns. Now we define q'_{rigid} as a disjunction whose first member slightly adapts q', while its second member π^- is the disjunction of all $\mathbf{\Pi}(\downarrow, \to)$ patterns extending π with one single node labeled with wildcard and with a fresh variable over data values. The key idea is now that if a complete tree t does not satisfy π^- but satisfies q_{rigid} via some homomorphism h, then for every x_i, either $h(x_i) = 0$, or $h(x_i) = 1$, i.e., t encodes one particular valuation of the p_is. □

Since wildcard can lead to an increase in complexity of the containment problem, it is natural to ask then when we can match the previously established complexity results in the presence of wildcard. For $\mathbf{\Pi}(\downarrow)$ and $\mathbf{\Pi}(\downarrow, \to)$ patterns the answer to this is surprisingly simple: we can allow wildcard everywhere except at the root of the pattern. Recall that in Section 5 we associated with each pattern π an incomplete tree t_π with a unique root. The requirement is basically that the label of the root of t_π is $a \in \mathcal{L}$; other nodes of t_π can be labeled either by $a \in \mathcal{L}$ or by $\_$.

For instance, the following rules define such patterns based on child-only navigation:

$$\begin{array}{rcll} \pi & := & a(x)[\pi', \ldots, \pi'] & a \in \mathcal{L} \\ \pi' & := & a(x)[\pi', \ldots, \pi'] & a \in \mathcal{L} \cup \{\_\} \end{array} \quad (8)$$

That is, the π's define patterns that can use wildcard, and π is the top-level pattern, whose root label comes from $\mathcal{L}$.

When we have this restriction on patterns with wildcard, we write $\mathbf{\Pi}(\sigma, \_^{\neg r})$, where σ, as before, is a set of axes. Likewise we define classes of queries – e.g., $\mathrm{CQ}(\downarrow, \to, \_^{\neg r})$ – and containment problems – e.g., $\mathrm{CQ}_{\subseteq}(\downarrow, \to, \_^{\neg r})$.

Obviously the addition of wildcard preserves lower bounds. We have already seen that containment of BCCQs with wildcard is in Π_2^p, and hence all three versions of BCCQ containment – $\mathrm{BCCQ}_{\subseteq}(\Downarrow, \Rightarrow)$, $\mathrm{BCCQ}_{\subseteq}(\Downarrow, \Rightarrow, \_)$, and $\mathrm{BCCQ}_{\subseteq}(\Downarrow, \Rightarrow, \_^{\neg r})$ – are Π_2^p-complete.

Now we show that the NP bounds established via homomorphisms are also preserved when wildcard is used everywhere except the root.

PROPOSITION 6.3. *The problems* $CQ_{\subseteq}(\downarrow, \_^{\neg r})$ *and* $CQ_{\subseteq}(\downarrow, \to, \_^{\neg r})$ *are* NP*-complete.*

Proof sketch. We adapt the proof of the corresponding result in Section 5. We now turn t_Q into a complete tree using a slightly different procedure. We just add to it one new root node labeled $\heartsuit(\sharp)$ and we decide arbitrarily on a sibling ordering when none is specified. We finally substitute fresh distinct constants for every distinct variable, thus obtaining a complete tree T. The remainder of the proof is almost as before. Whenever the next sibling relation is available, we only need to notice that the homomorphism $h_2 : t_{Q'} \to T$ cannot map any node in $t_{Q'}$ to the root of T. As tree patterns are rooted, this entails that nothing can be said in $t_{Q'}$ about the relative sibling orderings of the preimages of the children of the root of T. □

Note that such a procedure would not work when both unions of siblings and next sibling are allowed. For instance, let $q = \exists x, y, z\ a(x)[a(y), b(z)]$ and $q' = \exists x, y, z\ a(x)[\_(y) \to \_(z)]$ with $a \neq b$. Obviously $q \subseteq q'$, as q forces the tree to have an a-labeled node with at least two children. On the other hand, it is easy to see that there is no homomorphism from $t_{q'}$ to t_q.

Similarly, the method cannot be applied to queries in

$\mathrm{CQ}(\Downarrow, \_^{\neg r})$. Consider $q = \exists x, y\ a(x)//b(y)$ and $q' = \exists x, y, z\ a(x)/\_(z)//b(y) \wedge \exists x, y, z\ a(x)//\_(z)/b(y)$, with $a \neq b$. Again $q \subseteq q'$, as q forces the tree to have an a-labeled node which has at least one child and a b-labeled descendant which has a parent. But here again, it is obvious that there is no homomorphism from $t_{q'}$ to $(t_q)^*$.

Observe finally that by allowing wildcard to appear everywhere in patterns we also lose the homomorphism criterion that let us establish the NP upper bound. For instance, let $q = \exists x, y\ \big(a(x) \wedge b(y)\big)$, with $a \neq b$, and let $q' = \exists x, y\ (\_(x)/\_(y))$. Since q forces each tree to have at least two nodes, we have the containment $q \subseteq q'$; however there is no homomorphism from $t_{q'}$ to t_q.

As the last result of this section, we show that combining unions of queries even with the restricted use of wildcard can increase the complexity of containment.

PROPOSITION 6.4. *Containment of UCQs that use downward navigation and wildcard except at the root, i.e., the problem* $\mathrm{UCQ}_{\subseteq}(\Downarrow, \_^{\neg r})$*, is* Π_2^p*-complete.*

Proof sketch. We adapt the proof of Theorem 4.2 along the same lines as in the proof of Theorem 6.2. We define queries $q_{\Downarrow}, q'_{\Downarrow}$ as follows. We keep all the $\downarrow$ paths patterns which were actually used in q to encode the clauses of φ, but we now encode the valuation of the p_i's using a pattern $\pi_1/\ldots/\pi_l$ where for each $1 \leq i \leq l$, $\pi_i = \mathit{Val}(0)//\mathit{Val}(x_i)//\mathit{Val}(1)$. We can now construct $q'_{\Downarrow}$ almost as in the proof of Theorem 6.2, except that we replace π_- with a CQ $\exists x_1 \ldots \exists x_{2l+1} \mathit{Val}(0)/\_(x_1)/\ldots/\_(x_{2l+1})$. □

7. THE EFFECT OF SCHEMAS

So far we have not assumed any schema information, such as a DTD or a more general schema description, under which we perform static analysis of queries. However, such assumptions are fairly common, as many XML documents are required to satisfy schema descriptions. Schemas are very well known to affect static analysis of XML. In fact containment of queries can easily behave differently under schemas, even such simple ones as specifying the label of the root of a document. For instance, if $q = \exists x, y\ \big(a(x) \wedge b(y)\big)$ and $q' = \exists x, y\ \big(c(x)/\_(y)\big)$, then in general $q \not\subseteq q'$, but if we state that roots must be labeled c, then $q \subseteq q'$.

In addition, the presence of schemas is known to affect the complexity of static reasoning tasks, generally by increasing it, sometimes even making it undecidable [7, 10, 17, 18, 21, 33, 35]. The main observation of this section is that under schema information, we preserve decidability of query containment for those classes we have encountered so far, but at the cost of an exponential blow-up.

Abstraction of XML schemas. There are many formalisms for describing XML schemas (see, e.g., [29] for a survey), but most of them are subsumed by the notion of an unranked tree automaton. To define it, fix a finite alphabet $\Sigma \subset \mathcal{L}$. A *non-deterministic unranked tree automaton (NTA)* [32, 36] over Σ-labeled trees is a tuple $\mathcal{A} = (Q, \Sigma, \delta, F)$, where Q is a finite set of states, $F \subseteq Q$ is the set of final states, and $\delta : Q \times \Sigma \to 2^{(Q^*)}$ is a transition function. We require that the $\delta(q, a)$'s be regular languages over Q for all $q \in Q$ and $a \in \Sigma$. When we deal with complexity results involving automata, we assume that these regular languages are represented by NFAs (or by regular expressions, since those can be converted into NFAs in polynomial time).

A run of $\mathcal{A}$ over a tree t with domain D and labeling function λ is a function $r_{\mathcal{A}} : D \to Q$ such that for each node s with n children $s \cdot 0, \ldots, s \cdot (n-1)$, the word $r_{\mathcal{A}}(s \cdot 0) \cdots r_{\mathcal{A}}(s \cdot (n-1))$ is in the language $\delta(r_{\mathcal{A}}(s), \lambda(s))$. So, for a leaf s labeled a this means that s could be assigned state q iff the empty word ϵ is in $\delta(q, a)$. A run is accepting on tree t if the root of t is assigned an accepting state (formally, $r_{\mathcal{A}}(\epsilon) \in F$. A tree t is accepted by $\mathcal{A}$ if there exists an accepting run of $\mathcal{A}$ on t. The set of all trees accepted by $\mathcal{A}$ is denoted by $\mathcal{L}(\mathcal{A})$.

We then define the containment problem under schemas as follows. Let $\mathcal{Q}$ be one of the classes CQ, UCQ, or BCCQ, and σ a set of axes.

PROBLEM:	$\mathcal{Q}_{\subseteq}(\sigma)$ under schemas
INPUT:	queries $q(\bar{x}), q'(\bar{x}')$ in $\mathcal{Q}(\sigma)$ and NTA $\mathcal{A}$;
QUESTION:	is $q(t) \subseteq q'(t)$ for every $t \in \mathcal{L}(\mathcal{A})$?

A general upper bound. We show that all the versions of $\mathcal{Q}_{\subseteq}(\sigma)$ remain decidable under schemas, but the upper bound is one exponent higher than it was without schemas.

THEOREM 7.1. $\mathrm{BCCQ}_{\subseteq}(\Downarrow, \Rightarrow, \_)$ *under schemas is* 2EXPTIME*-complete.*

Proof sketch. The idea is to prove that we can reduce $\mathrm{BCCQ}_{\subseteq}(\Downarrow, \Rightarrow, \_)$ under schemas to a similar problem over finite alphabets and that we can encode a $\mathrm{CQ}(\Downarrow, \Rightarrow, \_)$ into an exponential-size unranked tree automaton. The 2EXPTIME upper bound then follows from tree automata techniques. The lower bound is immediate from Theorem 7.2.

Lower bounds. Since $\mathrm{BCCQ}_{\subseteq}(\Downarrow, \Rightarrow, \_)$ without the presence of schemas is in Π_2^p (and therefore in single-exponential time), it is natural to ask to whether the jump to double-exponential time is unavoidable. It turns out that it is, even for conjunctive queries, as we can prove the following.

THEOREM 7.2. $\mathrm{CQ}_{\subseteq}(\Downarrow, \Rightarrow)$ *under schemas is* 2EXPTIME*-complete.*

Proof sketch. The upper bound is immediate from Theorem 7.1. The lower bound is obtained in two steps. First we show that we can transfer lower bounds for $\mathrm{UCQ}_{\subseteq}(\Downarrow, \Rightarrow)$ under schemas to lower bounds for $\mathrm{CQ}_{\subseteq}(\Downarrow, \Rightarrow)$ under schemas by adapting a technique from [31]. Then we prove the lower bound for $\mathrm{UCQ}_{\subseteq}(\Downarrow, \Rightarrow)$ by a reduction from the acceptance problem for alternating exponential space bounded Turing machines. This is done by adapting the

proof of the 2EXPTIME lower bound for query containment from Theorem 6 in [10]. Two difficulties arise as that proof used queries with node equalities and wildcards. We handle node equalities by using data value equality constraints in our setting. We show how we can enforce all nodes from a tree to have different data values and then we simulate node equality by data equality. We further provide a modification of the encoding that avoids the use of wildcard. □

We do not yet have a complete classification of what happens for all of the classes of queries under schemas, but we do have an indication very little is needed to make their complexity considerably higher than in the schema-less scenario. In fact one can use results from [12] to prove that even for very simple classes of queries (child relation only; no branching), containment under schemas *provably* requires exponential time.

8. THE EFFECT OF DATA VALUE COMPARISONS

The last feature we are going to consider is data value comparisons, specifically disequalities $\neq$. This is a standard addition that has been considered in the study of relational conjunctive queries. In fact it is one of the mildest ways of adding a limited form of negation to positive queries in a way that preserves their nice properties, such as the decidability of static analysis. The other such extension, also considered here, is allowing Boolean combinations of CQs.

The relational case of CQs with $\neq$ comparisons has been settled in [25, 26, 37]: the containment problem is Π_2^p-complete. From this we can derive some hardness results, for instance, containment of $\mathrm{CQ}(\downarrow)$ with disequalities under schema is Π_2^p-hard (note that the schema assumption is necessary here to ensure documents code relational databases, as was already explained in Section 4). As for upper bounds, for relational BCCQs, even with disequalities, containment is decidable. In fact it is easily seen that such containment reduces to the complement of satisfiability for the Bernays-Schönfinkel class.

However, relational results do not give us any *upper* bounds on the containment problem for XML queries. We show in this section that there is a reason for it: such problems are, by and large, *undecidable*. In fact we show two undecidability results: for XML BCCQs with data comparisons, and even for CQs in the presence of schema information.

Queries with data comparisons. We now formally define classes of queries with $=$ and $\neq$ data comparisons. Suppose we start with a class $\mathbf{\Pi}(\sigma)$ of patterns. Then *CQs with data comparisons* over σ are defined as

$$q(\bar{x}) \;=\; \exists \bar{y}\,\big(\bigwedge_{i=1}^{n} \pi_i(\bar{z}_i) \;\wedge\; \alpha(\bar{x},\bar{y})\big), \qquad (9)$$

where all the π_is are patterns from $\mathbf{\Pi}(\sigma)$ and α is a conjunction of formulae of the form $u = v$ and $u \neq v$, where the variables u and v come from $\bar{x}$ and $\bar{y}$. For instance, $q(x) = \exists y\,(a[b(x), c(y)] \wedge x \neq y)$ is such a query.

The class of such queries will be denoted by $\mathrm{CQ}(\sigma, \sim)$ (using the common XML literature notation of $\sim$ for data value comparisons). We then define the class $\mathrm{UCQ}(\sigma, \sim)$ as unions of queries in $\mathrm{CQ}(\sigma, \sim)$, and $\mathrm{BCCQ}(\sigma, \sim)$ as Boolean combinations of such queries.

Before we present our results, notice that in (9), the formula α allows explicit equalities. Normally in CQs these can be avoided simply by collapsing two variables. However, in the case of pattern-based queries, we may actually need explicit equalities, at least for UCQs. Consider, for example, a Boolean query $q(x, y) = a(x) \wedge a(y)$. Then this query implies the following UCQ $q'(x, y) = (x = y) \vee \_/\_$. Indeed, if $q(x, y)$ is witnessed by two data values that are different, then they must occur in different nodes and hence the $\_/\_$ pattern is true.

Containment without schemas. Without schemas, the containment problem for BCCQs behaves drastically differently from the relation case, as we show below.

THEOREM 8.1. *Containment of BCCQs with data comparisons, i.e., the problem* $\mathrm{BCCQ}_{\subseteq}(\Downarrow, \Rightarrow, \_, \sim)$*, is undecidable.*

In fact one needs either $\downarrow, \downarrow^*, \rightarrow$ or $\downarrow, \rightarrow, \rightarrow^*$ to establish undecidability.

Proof sketch. The proof shows that satisfiability for a BCCQ is undecidable by reduction from Post's Correspondence Problem (PCP). The proof is rather technical. It may be tempting to think that, since $\mathrm{BCCQ}(\Downarrow, \Rightarrow, \_, \sim)$ can express certain key constraints, one can simulate the node equality tests from [10] in our setting by data equalities, and then we can adapt undecidability results from there as well. However, under such a key constraint, it is not clear at all how then the data equalities and inequalities from [10] can be correctly simulated. The reduction from PCP consists of a series of encoding steps that state that (1) all trees satisfying the BCCQ must be string-shaped and of a certain form; and (2) that they somehow encode a PCP solution. The proof can be done in two flavors: either we say that the tree does not branch, in which case we need the negation of the $\rightarrow$ predicate to express (1) as well as both $\downarrow$ and $\downarrow^*$ for (2). Alternatively, we say that the root has no grandchildren, in which case we need $\downarrow$ for (1) and $\rightarrow$ and $\rightarrow^*$ for (2). □

Containment with schemas. As in the previous section, for each containment problem of the form $\mathcal{Q}_{\subseteq}(\sigma, \sim)$, with $\mathcal{Q}$ being CQ, or UCQ, or BCCQ, we can associate an analogous containment problem *under schemas* which, in addition, will take as an input a schema, represented as an automaton.

The combination of data value comparisons and schemas has an even more severe effect on the complexity of the containment problem: it becomes undecidable already for CQs using only downward navigation.

THEOREM 8.2. *The containment problem for* $\mathrm{CQ}(\Downarrow, \sim)$ *queries under schema is undecidable.*

Proof sketch. As in the proof of Theorem 7.2, we first notice that we can transfer lower bounds for $\mathrm{UCQ}_{\subseteq}(\Downarrow, \sim)$ under schemas to lower bounds for $\mathrm{CQ}_{\subseteq}(\Downarrow, \sim)$. After that we prove undecidability for $\mathrm{UCQ}_{\subseteq}(\Downarrow, \sim)$ by reduction from the halting problem of two-counter machines. □

We conclude with the following remark. We noticed earlier that relational results give us Π_2^p-hardness for containment of $\mathrm{CQ}(\downarrow, \sim)$ queries under schemas (to enforce relational encoding). While the precise complexity of the problem $\mathrm{CQ}_{\subseteq}(\downarrow, \sim)$ remains open (see concluding remarks), we can at least eliminate the need for schemas from the hardness result, i.e., we can prove the following.

PROPOSITION 8.3. *The problem $CQ_{\subseteq}(\downarrow, \sim)$ is Π_2^p-hard.*

9. CONCLUSION

We have analyzed the containment problem for three classes of queries – CQs, UCQs, and BCCQs – based on various classes of tree patterns (including $\mathbf{\Pi}(\downarrow)$, $\mathbf{\Pi}(\downarrow, \rightarrow)$, $\mathbf{\Pi}(\Downarrow)$, and $\mathbf{\Pi}(\Downarrow, \Rightarrow)$), also in the presence of extra features such as wildcard, schemas, and disequality comparisons.

Overall, this gives us 96 cases of possible variations of the containment problem, and our results, although not generating the full set of 96 complexity bounds, have provided answers to the majority of them. Nonetheless, there are a few questions left open, that we would like to address. These concern the cases when we have some of the extra features (wildcard, schemas, inequalities) present.

With wildcard, without any restrictions, we do not yet have the precise complexity of containment for four classes: $\mathrm{CQ}(\downarrow, \_), \mathrm{CQ}(\downarrow, \rightarrow, \_)$, $\mathrm{CQ}(\Downarrow, \_)$, and $\mathrm{UCQ}(\downarrow, \_)$. With schemas, we do not yet know whether containment for CQs and UCQs without transitive closure axes is single-exponential or double-exponential. And with disequality comparisons, we do not know if containment without transitive closure axes is decidable. Based on our investigations, all these problems appear to be rather nontrivial. We plan to address them in the future.

Acknowledgment. This work was supported by the FET-Open project FoX (Foundations of XML), grant agreement FP7-ICT-233599, by EPSRC grant G049165, and by DFG grant MA 4938/2-1.

10. REFERENCES

[1] S. Abiteboul, B. Cautis, T. Milo. Reasoning about XML update constraints. In *PODS'07*, pages 195–204.
[2] S. Abiteboul, L. Segoufin, and V. Vianu. Representing and querying XML with incomplete information. *ACM TODS*, 31(1):208–254, 2006.
[3] N. Alon, T. Milo, F. Neven, D. Suciu, V. Vianu. XML with data values: typechecking revisited. *JCSS* 66(4): 688-727 (2003).
[4] S. Amano, L. Libkin, F. Murlak. XML schema mappings. In *PODS'09*, pages 33–42.
[5] S. Amer-Yahia, S. Cho, L. Lakshmanan, D. Srivastava. Tree pattern query minimization. *VLDB J.* 11(4): 315-331 (2002).
[6] M. Arenas, P. Barceló, L. Libkin, F. Murlak. *Relational and XML Data Exchange*. Morgan & Claypool, 2010.
[7] M. Arenas, W. Fan, L. Libkin. On the complexity of verifying consistency of XML specifications. *SIAM J. Comput.* 38(3): 841-880 (2008).
[8] M. Arenas, L. Libkin. XML data exchange: consistency and query answering. *J. ACM* 55(2): (2008).
[9] P. Barceló, L. Libkin, A. Poggi, C. Sirangelo. XML with incomplete information. *J. ACM*, 58:1 (2010).
[10] H. Björklund, W. Martens, T. Schwentick. Optimizing conjunctive queries over trees using schema information. *MFCS'08*, pages 132–143.
[11] H. Björklund, W. Martens, and T. Schwentick. Conjunctive query containment over trees. *JCSS* 77(3): 450-472 (2011).
[12] H. Björklund, W. Martens, T. Schwentick. Validity of tree pattern queries with respect to schema information. Unpublished manuscript.
[13] M. Bojanczyk, C. David, A. Muscholl, T. Schwentick, L. Segoufin. Two-variable logic on words with data. In *LICS'06*, pages 7-16.
[14] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Regular XPath: constraints, query containment and view-based answering for XML documents. In *LID'08*.
[15] A. Chandra and P. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC 1977*, pages 77–90.
[16] C. David. Complexity of data tree patterns over XML documents. *MFCS'08*, pages 278–289.
[17] W. Fan, L. Libkin. On XML integrity constraints in the presence of DTDs. *J. ACM* 49(3): 368–406 (2002).
[18] D. Figueira. Satisfiability of downward XPath with data equality tests. *PODS'09*, 197-206.
[19] P. Genevés and N. Layaida. A system for the static analysis of XPath. *ACM TOIS* 24 (2006), 475–502.
[20] A. Gheerbrant, L. Libkin, and T. Tan. On the complexity of query answering over incomplete XML documents. *ICDT 2012*, 169–181.
[21] G. Gottlob, C. Koch, K. Schulz. Conjunctive queries over trees. *J. ACM* 53 (2006), 238–272.
[22] T. J. Green. Containment of conjunctive queries on annotated relations. *Theory Comput. Syst.* 49(2): 429-459 (2011).
[23] A. Halevy. Answering queries using views: A survey. *VLDB J.* 10(4):270-294 (2001).
[24] B. Kimelfeld, Y. Sagiv. Revisiting redundancy and minimization in an XPath fragment. *EDBT'08*, pages 61-72.
[25] A. Klug. On conjunctive queries containing inequalities. *J. ACM* 35(1): 146-160 (1988).
[26] P. Kolaitis, D. Martin, M. Thakur. On the complexity of the containment problem for conjunctive queries with built-in predicates. In *PODS 1998*, pages 197-204.
[27] M. Lenzerini. Data integration: a theoretical perspective. In *PODS'02*, pages 233–246.
[28] L. Libkin, C. Sirangelo. Reasoning about XML with temporal logics and automata. *J. Applied Logic*, 8:2, 210–232 (2010).
[29] W. Martens, F. Neven, T. Schwentick. Simple off the shelf abstractions for XML schema. *SIGMOD Record* 36(3): 15-22 (2007).
[30] S. Maneth, T. Perst, H. Seidl. Exact XML type checking in polynomial time. In *ICDT 2007*, pages 254–268.
[31] G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *J. ACM*, 51(1): 2–45, 2004.
[32] F. Neven. Automata, logic, and XML. In *CSL 2002*, pages 2–26.
[33] F. Neven, T. Schwentick. On the complexity of XPath containment in the presence of disjunction, DTDs, and variables. *LMCS*, 2(3): (2006).
[34] Y. Sagiv, M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *J. ACM* 27(4): 633-655 (1980).
[35] Th. Schwentick. XPath query containment. *SIGMOD Record* 33(1): 101-109 (2004).
[36] J.W. Thatcher. Characterizing derivation trees of context-free grammars through a generalization of finite automata theory. *JCSS* 1 (1967), 317–322.
[37] R. van der Meyden. The complexity of querying indefinite data about linearly ordered domains. *JCSS* 54(1): 113-135 (1997).
[38] V. Vianu. A web Odyssey: from Codd to XML. In *PODS'01*, pages 1–15.

APPENDIX

A. THE PROOF FROM SECTION 3

In this section we add a single result, namely the Π_2^p upper bound for the containment of BCCQs that use all the axes. We actually show a more general upper bound for queries that also use wildcard, i.e., we prove that the problem $\mathrm{BCCQ}_{\subseteq}(\Downarrow, \Rightarrow, \_)$ is in Π_2^p.

We assume for now that a query is Boolean, i.e. does not have free variables. We do so for keeping the notation simple. The proof with free variables is essentially the same, and we shall explain the minor changes that need to be made to incorporate free variables at the end.

Note that $q \subseteq q'$ holds iff $q - q'$ always return the empty set (i.e., is not satisfiable in the Boolean case). Since $q - q'$ is a BCCQ if so are q and q', all we need to do is to provide a Σ_2^p algorithm for checking satisfiability of $\mathrm{BCCQ}(\Downarrow, \Rightarrow, \_)$ queries. Towards such an algorithm, assume that we are given a query q which is a Boolean combination of queries $q_1, \ldots, q_m$. For it to be satisfiable, we must have a function $\chi : \{1, \ldots, m\} \to \{0, 1\}$ and a tree t so that:

1. $t \models q_i$ for each i with $\chi(i) = 1$, and
2. $t \models \neg q_j$ for each j with $\chi(j) = 0$, and
3. setting each q_i with $\chi(i) = 1$ to true and q_j with $\chi(j) = 0$ to false makes the Boolean combination evaluate to true.

This function χ will actually become a part of the existential guess in our Σ_2^p algorithm. Let

$$q_\chi \;=\; \bigvee_{\chi(j)=0} q_j.$$

Then item 2 above can be restated as $t \models \neg q_\chi$, where q_χ is a UCQ. So our problem boils down to solving the following problem that we call $\mathrm{SAT}(\mathcal{Q}, q')$:

- given a collection $\mathcal{Q}$ of CQs, and a UCQ q', is there a tree t that satisfies every query in $\mathcal{Q}$ and does not satisfy q'?

Upon a correct guess of χ, we would just need to solve this problem for $\mathcal{Q} = \{q_i \mid \chi(i) = 1\}$ and $q' = q_\chi$.

We now recast $\mathrm{SAT}(\mathcal{Q}, q')$ as the problem of computing certain answers in XML data exchange [8], to apply (or, rather, adapt) upper bounds established there. For each query $q'' \in \mathcal{Q}$, define a source-to-target constraint $r \to q''$, where r is a label, and let $\Sigma_\mathcal{Q}$ be the set of all such constraints. We assume that schemas in data exchange come with no restrictions whatsoever (i.e., they may be given by fixed automata accepting all trees; the upper bounds of [8] apply in such a case). Recall that certain answers in data exchange, $\mathit{certain}_\Sigma(t, q)$ (for a Boolean query q) return true if q is true in every target tree, for a given source t and mapping Σ. This, in particular, tells us, that for a tree t_r consisting of a single node labeled r, we have:

- $\mathit{certain}_{\Sigma_\mathcal{Q}}(t_r, q')$ returns false iff $\mathrm{SAT}(\mathcal{Q}, q')$ returns true.

So we reduced the problem to checking certain answers in XML data exchange. The problem was studied with respect to data complexity, which per se is not of interest to us, as the source tree is fixed. However, by analyzing the proof in [8] one can observe the following. If there is a target tree t such that $q'(t)$ is false (i.e, if $\mathit{certain}_{\Sigma_\mathcal{Q}}(t_r, q')$ returns false), then there is such a target tree t_0 satisfying the following conditions:

- the size of t_0 is polynomial in t_r and $\mathcal{Q}$;
- the size of t_0 is $2^{O(|q'|)}$, where $|q'|$ measures the size of q' (technically, for these two items, what we mean is that there is a concrete polynomial p and a concrete linear function f so that the size of t_0 is bounded by $p(|t_r| + |\mathcal{Q}|)$ as well as by $2^{f(|q'|)}$; these functions are calculated in [8]);
- the only witnesses to the exponential size of t_0 (in $|q'|$) are two types of paths:
  - a vertical path between two fixed nodes such that the degree of branching for each internal node of the path is 1 (i.e., there is no branching), and such that each of the dependencies $r \to q''$ can be witnessed outside such a path (i.e., the image of the homomorphism from the tree representation of q'' into t_0 has empty intersection with such a path); and
  - a horizontal path between two fixed nodes such that each internal node of the path is a leaf, and such that again each of the dependencies $r \to q''$ can be witnessed outside such a path (we remark that the proof, as given in [8], only talks about vertical paths, but adding horizontal paths changes nothing at all, see, e.g., [9]).

In [8], where data complexity was considered, this was sufficient to prove that falsity of certain answers is in NP. Now we need to deal with combined complexity, so we cannot guess a tree with exponentially long paths. But we can do two things.

First, we pick a label $\ell \in \mathcal{L}$ that is not present in any of the queries, and relabel internal nodes of those exponential paths by ℓ. Furthermore, we assign to them distinct data values that are not used elsewhere in t_0 and not used as constants in the queries in $\mathcal{Q}$ and in q' (in particular, no two such nodes would have the same data value). Let the resulting tree be t'_0. It then immediately follows that t'_0 still satisfies all the q''s from $\mathcal{Q}$, and still does not satisfy q', as the changes could not turn any pattern from $\mathcal{Q}$ into satisfiable one, and all the q''s were witnessed outside of exponential paths.

Second, we take t'_0, and turn it into a polynomial-size data structure called *code*(t'_0) as follows. Let N be the set of special nodes in t'_0, i.e., endpoints of the exponential-length paths. Then *code*(t'_0) contains all the information about t'_0 except it does not keep the exponential paths. However, for every two special nodes n, n' it records the following: whether n' is a descendant or a following-sibling of n such that there is no other special node on the unique path between (i.e., they are consecutive special nodes, in the vertical or horizontal ordering), and in that case, the length of the path between n and n', encoded in binary. Given the bounds on the paths, we need at most $O(|q'|)$ bits to encode the lengths of such paths, so the entire *code*(t'_0) now becomes polynomial in q'.

We next need to show that for each CQ q_0, we can use *code*(t'_0) to verify whether $t'_0 \models q_0$, and the complexity of this checking is still in NP, even with exponentially more succinct representation. Normally one would check for the existence of a homomorphism from t_{q_0}, the tableau of q_0, into t'_0. Instead, we define a *semi-homomorphism* $h : t_{q_0} \to$ *code*(t'_0) just as a homomorphism, except that each node of t_{q_0} is mapped into either:

- a node of *code*(t'_0), or
- a pair of special nodes n, n' such that n' is either a descendant of n or a following sibling of n with no other special nodes between them, and a number k, represented in binary, with at most $O(|q_0|)$ bits.

Such a map h naturally gives rise to a map $h' : t_{q_0} \to t'_0$: in the second case, the node is mapped by h' into the kth successor of n on the unique – vertical or horizontal – path from n to n'. We then call h a semi-homomorphism iff the map h' is a usual homomorphism from t_{q_0} to t'_0. A key observation is that for a map $h : t_{q_0} \to$ *code*(t'_0) one can check if it is a semi-homomorphism in polynomial time. Indeed, the information on nodes and offsets is sufficient for checking all the relations $\downarrow, \rightarrow, \downarrow^*, \rightarrow^*$, and since data values on the exponential paths have been changed, we know that any data value on such a path is different from any other data value in the document.

To sum up, for checking whether *certain*$_{\Sigma_\mathcal{Q}}(t_r, q')$ returns false, it suffices to find a counterexample t'_0 which can be encoded by a polynomial-size data structure *code*(t'_0), and then for all q_js comprising q' check that there is *no* semi-homomorphism from t_{q_j} into *code*(t'_0). At the same time, we have to check that there is a semi-homomorphism from $t_{q''}$ to *code*(t'_0) for each $q'' \in \mathcal{Q}$, to ensure that t'_0 is a solution.

Putting this together, we have a Σ_2^p algorithm for BCCQ satisfiability:

1. In the existential step, we guess:
   - a map $\chi : \{1, \ldots, m\} \to \{0, 1\}$;
   - a data structure S of the form *code*(t'_0) of polynomial size;
   - semi-homomorphisms $h_i : t_{q_i} \to S$ for $\chi(i) = 1$.
2. In the universal step, we consider all
   - semi-homomorphisms $g_j : t_{q_j} \to S$ for $\chi(j) = 0$.
3. Then, in polynomial time, we check:
   - that S is of the form *code*(t'_0) (that is, it is a tree structure, and an offset is associated with every pair of consecutive special nodes in the horizontal or vertical ordering);
   - that all the h_is and g_js are semi-homomorphisms (which we know can be done in polynomial time).

This completes the description of the algorithm in the Boolean case. With free variables $\bar{x} = (x_1, \ldots, x_n)$ in the query, we take just a few small changes. We shall also need to guess a tuple $\bar{v}$ satisfying a query. Its data values can in general be arbitrary, so the only thing we guess is the equality pattern with respect to the constants used in the query (i.e., we guess which free variables are assigned the same data values, and which ones are assigned constants used in queries). For instance, suppose the queries mentioned constants $1, \ldots, k$, and we have free variables x, y, z, u. Then an example of an equality pattern that we may guess is that $x = y$ while z and u is different, and $z = 1$, while other variables are assigned values not present in queries. Then we shall assign values $k + 1$ to x and y, 1 to z, and $k + 2$ to u to satisfy the equality pattern. Note that the values that are not constants present in queries can be arbitrary, but we can always choose them so that the size of the guess is polynomial (in fact, linear). This guess will be added to the existential guessing stage in the algorithm. The constraints in

the data exchange setting will be changed to $r[a(x_1) \to a(x_2) \to \ldots \to a(x_n)] \to q''(x_1, \ldots, x_n)$. That is, the input tree has root r and n children labeled a, holding values of the free variables. Finally, given the guess of equality patterns, and a corresponding tuple of values $\bar{v} = (v_1, \ldots, v_n)$, the input tree will be $r[a(v_1) \to a(v_2) \to \ldots \to a(v_n)]$. The rest of the proof then applies verbatim. This completes the proof of Theorem 3.1 and Proposition 6.1, since all the bounds we used apply in the case of wildcard.

B. PROOFS FROM SECTION 4

Theorem 4.1. *The problem* $\mathrm{BCCQ}_{\subseteq}(\downarrow)$ *is* Π_2^p*-complete.*

PROOF. We proceed by reduction from $\forall\exists 3CNF$. An instance of this problem is given as follows by a fully quantified Boolean formula φ in prenex conjunctive normal form:

$$\varphi := \forall p_1 \ldots \forall p_l \exists p_{l+1} \ldots \exists p_m \bigwedge_{1 \leq i \leq n} (l_{i1} \vee l_{i2} \vee l_{i3}),$$

where each l_{ij} is a literal over the p_i's (i.e., an atom or a negation of atom). Given as input such a formula φ, the problem of deciding whether there exists for each truth assignment of the $p_1, \ldots, p_l$'s, a truth assignment of the $p_{l+1}, \ldots, p_m$'s which makes the Boolean formula $\bigwedge_{1 \leq i \leq n}(l_{i1} \vee l_{i2} \vee l_{i3})$ true, is known to be Π_2^p-complete.

We follow the exact same strategy as in [34] and take full advantage of the fact that there are exactly three literals per clause in our input formulas. As the argument is just a straightforward adaptation of the one in [34], we only sketch it. Given such a formula

$$\varphi := \forall p_1 \ldots \forall p_l \exists p_{l+1} \ldots \exists p_m \bigwedge_{1 \leq i \leq n} (l_{i1} \vee l_{i2} \vee l_{i3}),$$

we use the alphabet $\{Cl, Lit\}$ and construct queries $q_{BCCQ}, q'_{BCCQ} \in \mathrm{BCCQ}(\downarrow)$ such that:

$$\varphi \quad \text{is true} \qquad \text{if and only if} \qquad q_{BCCQ} \subseteq q'_{BCCQ}.$$

We let:

$$q_{BCCQ} = \bigwedge_{1 \leq i \leq l} (\Pi_i^0 \vee \Pi_i^1)$$

$$q'_{BCCQ} = \exists x_1 \ldots \exists x_m \, \Pi_\varphi$$

We start by explaining how to construct Π_φ as a simple encoding of the quantifier-free part of the formula φ. To each propositional variable p_i occurring in φ we associate the variable x_i. Now if the i^{th} clause in φ is over the variables $\{p_j, p_k, p_l\}$ we code it (regardless of the fact that variables appear positively or negatively in it) using the following pattern:

$$Cl(c_i)/Lit(x_j)/Lit(x_k)/Lit(x_l)$$

We finally construct Π_φ as the conjunction of all the n patterns obtained in this way.

We now explain how to construct the disjunction $\Pi_i^0 \vee \Pi_i^1$, for each $1 \leq i \leq l$. First observe that for any clause containing three literals, there are seven valuations over the variables occurring in the clause which make the clause evaluates to true and only one which makes it evaluates to false. E.g., let $(p_1 \vee \neg p_2 \vee p_3)$ be the i^{th} clause of φ, the only valuation for which this clause evaluates to false is $p_1 = 0, p_2 = 1, p_3 = 0$. We want to represent all the valuations over the variables in the clause, but this one (as we are interested in φ being true). So if the i^{th} clause of φ is of the form $(p_1 \vee \neg p_2 \vee p_3)$, we encode it as the following conjunction of seven patterns:

$$Cl(c_i)/Lit(0)/Lit(0)/Lit(0)$$

$$\wedge$$

$$Cl(c_i)/Lit(1)/Lit(1)/Lit(1)$$

$$\wedge$$

$$Cl(c_i)/Lit(1)/Lit(1)/Lit(0)$$

$$\wedge$$

$$Cl(c_i)/Lit(0)/Lit(0)/Lit(1)$$

$$\wedge$$

$$Cl(c_i)/Lit(0)/Lit(1)/Lit(1)$$

$$\wedge$$

$$Cl(c_i)/Lit(1)/Lit(0)/Lit(1)$$

$$\wedge$$

$$Cl(c_i)/Lit(0)/Lit(0)/Lit(1)$$

We now define Π_{all} as the set, for every clause in φ, of all the so-obtained patterns. Finally, for each $1 \leq i \leq l$, Π_i^0 and Π_i^1 are obtained from the two component partition of Π_{all}, where the first component contains the patterns representing the valuations where the variable p_i has been set to false, while the second component contains the patterns representing the valuations where the variable p_i has been set to true. We construct Π_i^0 as the conjunction of all the patterns in the first component of the partition and Π_i^1 as the conjunction of all the patterns in its second component.

To illustrate the construction let

$$\varphi = \forall p_1 \exists p_2 \exists p_3 (p_1 \vee \neg p_2 \vee p_3)$$

We obtain

$$\begin{aligned} q_{BCCQ} = \quad & \big(Cl(c_1)/Lit(0)/Lit(0)/Lit(0) \\ & \wedge \\ & Cl(c_1)/Lit(0)/Lit(0)/Lit(1) \\ & \wedge \\ & Cl(c_1)/Lit(0)/Lit(1)/Lit(1) \\ & \wedge \\ & Cl(c_1)/Lit(0)/Lit(0)/Lit(1)\big) \end{aligned} \quad \vee \quad \begin{aligned} & \big(Cl(c_1)/Lit(1)/Lit(0)/Lit(1) \\ & \wedge \\ & Cl(c_1)/Lit(1)/Lit(1)/Lit(1) \\ & \wedge \\ & Cl(c_1)/Lit(1)/Lit(1)/Lit(0)\big) \end{aligned}$$

$$q'_{BCCQ} = \quad \exists x_1 \exists x_2 \exists x_3 \; Cl(c_1)/Lit(x_1)/Lit(x_2)/Lit(x_3)$$

Notice that φ is a true formula and that indeed $q_{BCCQ} \subseteq q'_{BCCQ}$.

We now show that for each $\forall\exists 3CNF$-instance

$$\varphi := \forall p_1 \ldots \forall p_l \exists p_{l+1} \ldots \exists p_m \bigwedge_{1 \leq i \leq n} (l_{i1} \vee l_{i2} \vee l_{i3}),$$

$$\varphi \quad \text{is true} \qquad \text{if and only if} \qquad q_{BCCQ} \subseteq q'_{BCCQ}.$$

For the first direction, we assume φ is true and show that $\bigwedge_{1 \leq i \leq l} (\Pi_i^0 \vee \Pi_i^1) \subseteq \exists x_1 \ldots \exists x_m \; \Pi_\varphi$. So let T be a data tree with $T \models \bigwedge_{1 \leq i \leq l} (\Pi_i^0 \vee \Pi_i^1)$. It follows that there is at least one mapping $\sigma : \{1, \ldots, l\} \to \{0, 1\}$ with

$$T \models \bigwedge_{1 \leq i \leq l} \Pi_i^{\sigma(i)}$$

Pick one such σ arbitrarily and consider the valuation of the variables $\{p_1, \ldots, p_l\}$ to Booleans given by $p_i = \sigma(i)$. As φ is true, σ can be extended to a valuation $\sigma^+ : \{1, \ldots, m\} \to \{0, 1\}$ such that the quantifier free part of φ is true when each variable p_i occurring in it is interpreted by $\sigma(i)$. By construction of $\bigwedge_{1 \leq i \leq l}(\Pi_i^0 \vee \Pi_i^1)$, this valuation is encoded by some of the tree patterns in $\bigwedge_{1 \leq i \leq l} \Pi_i^{\sigma(i)}$ and so there is a homomorphism $h_1 : t_{q'_{BCCQ}} \to t_{\bigwedge_{1 \leq i \leq l} \Pi_i^{\sigma(i)}}$ (as q'_{BCCQ} was constructed as an encoding of φ).[1] As $T \models \bigwedge_{1 \leq i \leq l} \Pi_i^{\sigma(i)}$, there is also a homomorphism $h_2 : t_{\bigwedge_{1 \leq i \leq l} \Pi_i^{\sigma(i)}} \to T$. Homomorphisms being preserved by composition, $h_1 \circ h_2 : t_{q'_{BCCQ}} \to T$ is also a homomorphism and so $T \models q'_{BCCQ}$.

For the other direction, assume $q_{BCCQ} \subseteq q'_{BCCQ}$ and let $\sigma : \{p_1, \ldots, p_l\} \to \{0, 1\}$ be some arbitrary assignment. We want to show that σ extends to some $\sigma^+ : \{p_1, \ldots, p_m\} \to \{0, 1\}$ which makes the quantifier free part of φ true. So let $T \models \bigwedge_{1 \leq i \leq l} \Pi_i^{\sigma(i)}$, then also $T \models \bigwedge_{1 \leq i \leq l}(\Pi_i^0 \vee \Pi_i^1)$ and as $q_{BCCQ} \subseteq q'_{BCCQ}$, also $T \models \exists x_1 \ldots \exists x_m \, \Pi_\varphi$. The values of the x_i's under which Π_φ evaluates to true give the desired σ^+ (i.e., we can set each $\sigma^+(p_i)$ (with $i \geq l$) to the value of x_i). □

Theorem 4.2. *The problem CQ$_\subseteq(\Downarrow, \Rightarrow)$ is Π_2^p-complete.*

PROOF. We also proceed by reduction from $\forall\exists 3CNF$. However, the idea behind the reduction is quite different from the one used in the proof of Theorem 4.1. Let us note also that we mainly restrict to $3CNF$ for notational convenience and that the reduction would also apply to any $\forall\exists CNF$-formula (i.e., where there is no constraint on the number of literals in a clause).

We show that for each $\forall\exists 3CNF$-instance

$$\varphi := \forall p_1 \ldots \forall p_l \exists q_1 \ldots \exists q_m \bigwedge_{1 \leq i \leq n} (l_{i1} \vee l_{i2} \vee l_{i3}),$$

there exist queries $q, q' \in \text{CQ}(\Downarrow, \Rightarrow)$ such that:

$$\varphi \quad \text{is true} \qquad \text{if and only if} \qquad q \subseteq q'.$$

We use the following alphabet of node labels:

$$\{Val, V, Cl, Lit, Oc, Q, P\}.$$

Now recall that l is the number of universally quantified propositional variables, m the number of existentially quantified propositional variables and n the number of clauses (or conjuncts) in φ, we let

$$q = \exists x_1 \ldots x_l (\bigwedge_{1 \leq i \leq n, 1 \leq j \leq 3} \Pi_{ij} \wedge \Pi_{Val})$$

and

$$q' = \exists y_1 \ldots \exists y_m (\bigwedge_{1 \leq i \leq m} Val(y_i)$$

$$\wedge$$

$$\bigwedge_{1 \leq j \leq n} \exists z_1 z_2 z_3 (Cl(c_j)/L(z_1)/P(z_2)/Oc(0)$$

[1] For the definition of incomplete tree t_q of a query q, see Section 5.

$$\wedge$$

$$V(2)/[Val(z_2) \rightarrow^* Val(0)]$$

$$\wedge$$

$$Cl(c_j)/L(z_1)/P(z_3)/Oc(1)$$

$$\wedge$$

$$V(2)/[Val(1) \rightarrow^* Val(z_3)]$$

$$\wedge$$

$$\bigwedge_{1 \leq k \leq m} Cl(c_j)/L(z_1)/Q(q_k)/Oc(y_k))$$

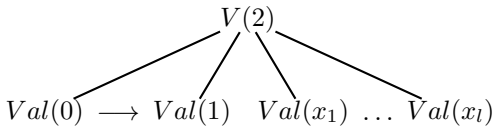
where the Π_{ij}'s and Π_{Val} are constructed as follows. For each literal l_{ij} in φ (where l_{ij} is the j^{th} literal of the i^{th} clause), we build Π_{ij} as follows:

1. Whenever the variable p_k occurring in l_{ij} is universally quantified in φ, we build Π_{ij} as the conjunction of the following patterns:
   - for every $1 \leq l \leq m$,
     $$Cl(c_i)/L(l_{ij})/Q(q_l)/Oc(0) \text{ and } Cl(c_i)/L(l_{ij})/Q(q_l)/Oc(1);$$
   - whenever $l_{ij} := p_k$,
     $$Cl(c_i)/L(l_{ij})/P(0)/Oc(0) \text{ and } Cl(c_i)/L(l_{ij})/P(x_k)/Oc(1);$$
   - whenever $l_{ij} := \neg p_k$,
     $$Cl(c_i)/L(l_{ij})/P(x_k)/Oc(0) \text{ and } Cl(c_i)/L(l_{ij})/P(1)/Oc(1).$$
2. Whenever the variable q_k occurring in l_{ij} is existentially quantified in φ, we build Π_{ij} as the conjunction of the following patterns:
   - $Cl(c_i)/L(l_{ij})/P(0)/Oc(0)$ and $Cl(c_i)/L(l_{ij})/P(1)/Oc(1)$;
   - whenever $l_{ij} := q_k$,
     $$Cl(c_i)/L(l_{ij})/Q(q_k)/Oc(1)$$
     and for every $1 \leq l \leq m$ such that $j \neq i$,
     $$Cl(c_i)/L(l_{ij})/Q(q_l)/Oc(0) \text{ and } Cl(c_i)/L(l_{ij})/Q(q_l)/Oc(1);$$
   - whenever $l_{ij} := \neg q_k$,
     $$Cl(c_i)/L(l_{ij})/Q(q_k)/Oc(0)$$
     and for every $1 \leq l \leq m$ such that $j \neq i$,
     $$Cl(c_i)/L(l_{ij})/Q(q_l)/Oc(0) \text{ and } Cl(c_i)/L(l_{ij})/Q(q_l)/Oc(1).$$

We finally construct Π_{Val} as the following pattern:

$$V(2)/[Val(0) \rightarrow Val(1), Val(x_1), \ldots, Val(x_l)]$$

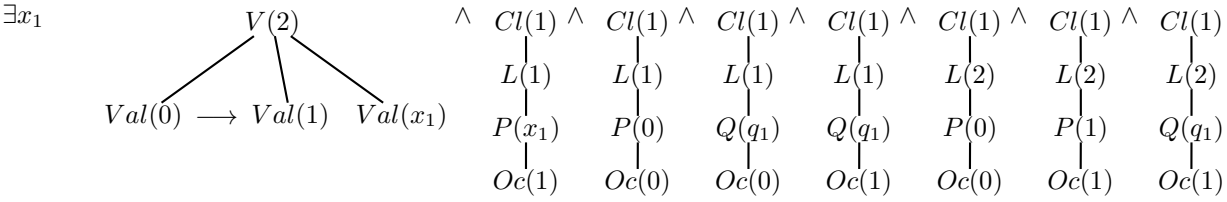
This pattern can also be seen as follows:



Observe that we slightly abused notations and that, strictly speaking, the query q' is not a conjunctive query as it contains nested existential quantification. However, it is easy to see that by renaming variables, q' can be put in prenex normal form

without increasing the size of the formula, i.e., it can be converted into another equivalent formula of the same size which is a conjunctive query.

Let us now give an example of our encoding applied to some specific input formula. As we already pointed out, the reduction actually also applies to any $\forall\exists CNF$-formula where clauses may contain some arbitrary number of literals. Our encoding being quite heavy, for clarity we will only show here how to encode the very simple true $\forall\exists CNF$-formula $\forall p_1 \exists q_1 (p_1 \vee q_1)$ containing one single clause with two literals. In this particular case, we can write the query q as follows:



In this simple case, one can easily verify that for every data tree T, if $T \models q$, then $T \models q'$. Indeed for any such tree T, there is a homomorphism $h : t_q \to T$ such that either $T \models V(2)[Val(h(x_1)) \to^* Val(0)]$, or $T \models V(2)[Val(1) \to^* Val(h(x_1))]$. But in both cases, $T \models q'$.

We now show that given some $\forall\exists 3CNF$-instance

$$\varphi := \forall p_1 \ldots \forall p_l \exists q_1 \ldots \exists q_m \bigwedge_{1 \leq i \leq n} (l_{i1} \vee l_{i2} \vee l_{i3})$$

$$\varphi \quad \text{is true} \qquad \text{if and only if} \qquad q \subseteq q'.$$

For the first direction assume φ is true. Now take some arbitrary data tree T such that $T \models q$. So there is a homomorphism $h : t_q \to T$ and T can naturally be associated to a valuation v of the p_i's such that $v(p_i) = 0$ whenever there is in T a $V(h(x_i))$-labeled node which is a child of a V(2)-labeled node and a left sibling of a $Val(0)$-labeled node and $v(p_i) = 1$ otherwise. Observe that in such a case, there is always a $V(h(x_i))$-labeled node which is a child of a V(2)-labeled node and a right sibling of a $Val(1)$-labeled node. As φ is true, the valuation v can be extended to a valuation v^+ of all the propositional variables in φ such that the quantifier-free part of φ is true whenever the p_i's and q_i's are interpreted according to v^+. We now show that $T \models q'$ by interpreting each existentially quantified variable y_i in q' by $v^+(q_i)$. Firstly, notice that the value assigned by v^+ to each q_i is either 0 or 1 and consequently for every $1 \leq i \leq m$, $T \models Val(v^+(q_i))$. Hence, the first big conjunction in q' holds for the values of the y_i's that we considered. Secondly, let us pick one j such that $1 \leq j \leq n$. Consider now the j^{th} clause of φ. By assumption there is a literal in this clause which is true for the valuation v^+. Assume it is the k^{th} literal. We will show that T satisfies the remaining conjuncts of q' where z_1 is interpreted by l_{jk} and each y_i is interpreted in T by $v^+(q_i)$.

There are two cases (we let l_{jk} stand for that literal):

1. the variable p_r in l_{jk} was universally quantified in φ:
   - $T \models \bigwedge_{1 \leq k \leq l}(Cl(c_j)/L(l_{jk})/Q(q_k)/Oc(v^+(q_k)))$ by construction of q and by the fact that h is a homomorphism, whatever the value of $v^+(q_i)$ is (that is, either 0 or 1),
   - whenever $l_{jk} := p_r$, $T \models Cl(c_j)/L(l_{jk})/P(z_2)/Oc(0) \wedge V(2)/[Val(z_2) \to^* Val(0)]$ holds by construction of q and by the fact that h is a homomorphism, with z_2 interpreted in T by 0. Also $T \models Cl(c_j)/lit(l_{jk})/P(z_3)/Oc(1) \wedge V(2)/[Val(1) \to^* Val(z_3)]$ because as $v^+(x_r) = 1$, this means that there is a $V(h(x_r))$-labeled node which is a child of a V(2)-labeled node and a right sibling of a $Val(1)$-labeled node, so we can take $h(x_r)$ as a value for z_3;
   - whenever $l_{jk} := \neg p_r$, $T \models Cl(c_j)/L(l_{jk})/P(z_3)/Oc(1) \wedge V(2)/[Val(1) \Rightarrow Val(z_3)]$ holds by construction of q and by the fact that h is a homomorphism, with z_3 interpreted in T by 1. Also $T \models Cl(c_j)/L(l_{jk})/P(z_2)/Oc(0) \wedge V(2)/[Val(z_2) \to^* Val(0)]$ because as $v^+(x_r) = 0$, this means that there is a $V(h(x_r))$-labeled node which is a child of a V(2)-labeled node and a left sibling of a $Val(0)$-labeled node, so we can take $h(x_r)$ as a value for z_2.
2. the variable q_r in l was existentially quantified in φ:
   - $T \models Cl(c_j)/L(l_{jk})/P(z_2)/Oc(0) \wedge V(2)/[Val(z_2) \to^* Val(0)] \wedge Cl(c_j)/L(l_{jk})/P(z_2)/Oc(1) \wedge V(2)/[Val(1) \to^* Val(z_3)] \wedge V(2)/[Val(1) \to^* Val(z_3)]$ holds with 0 as a value for z_2 and 1 as a value for z_3, by construction of Q and by the fact that h is a homomorphism;
   - $T \models \bigwedge_{1 \leq k \leq l}(Cl(c_j)/L(l_{jk})/Q(q_k)/Oc(v^+(q_k)))$ holds whatever are the values assigned to each $v^+(q_k)$, by construction of q and by the fact that h is a homomorphism. This holds for every q_i with $i \neq r$ because for every such i, $T \models Cl(c_j)/L(l_{jk})/Q(q_i)/Oc(0) \wedge Cl(c_j)/L(l_{jk})/Q(q_i)/Oc(1)$. Whenever $l_{jk} := q_r$, this holds in the case of q_r

because as $v^+(q_r) = 1$, we also assumed that x_r is evaluated by 1. Finally whenever $l_{jk} := \neg q_r$, this holds because as $v^+(q_r) = 0$, we also assumed that x_r is evaluated by 0.

Now for the converse direction assume the following formula to be false

$$\varphi := \forall p_1 \dots \forall p_l \exists q_1 \dots \exists q_m \bigwedge_{1 \leq i \leq n} (l_{i1} \vee l_{i2} \vee l_{i3})$$

I.e., the following formula is true:

$$\neg\varphi := \exists p_1 \dots \exists p_l \forall q_1 \dots \forall q_m \bigvee_{1 \leq i \leq n} (\neg l_{i1} \wedge \neg l_{i2} \wedge \neg l_{i3})$$

then there is a valuation v of the p_i's such that for every extension v^+ of v to the q_i's, $\bigvee_{1 \leq i \leq n}(\neg l_{i1} \wedge \neg l_{i2} \wedge \neg l_{i3})$ evaluates to true whenever the p_i's, q_i's are interpreted according to v^+. Now let $T \models q$ with $h : t_q \to T$ an onto homomorphism, where for every p_i, $h(x_i) = 0$ whenever $v(p_i) = 0$ and $h(x_i) = 1$ otherwise. Let also assume that T has only one single $V(2)$-labeled node which has only two children, the first one labeled $Val(0)$ and its next sibling labeled $Val(1)$. Assume $T \models q'$. As 0 and 1 are the only two data values d such that $T \models Val(d)$, there is a mapping σ of the existentially quantified variables $y_1, \dots, y_m$ to $\{0, 1\}$ such that

$$\begin{gathered} T \models \bigwedge_{1 \leq j \leq n} \exists z_1 z_2 z_3 (Cl(c_j)/L(z_1)/P(z_2)/Oc(0) \\ \wedge \\ V(2)/[Val(z_2) \to^* Val(0)] \\ \wedge \\ Cl(c_j)/L(z_1)/P(z_3)/Oc(1) \\ \wedge \\ V(2)/[Val(1) \to^* Val(z_3)] \\ \wedge \\ \bigwedge_{1 \leq k \leq l} Cl(d_j)/L(z_1)/Q(q_k)/Oc(\sigma(y_k))) \end{gathered}$$

This means that for each data value c_j occurring in T, there exists some k such that the remaining of the formula holds on T whenever z_1 is interpreted by l_{jk} (such l_{jk} values being the only ones to which we can set this existentially quantified variable). Now as σ is a mapping to Boolean values, there exists an extension v^+ of v such that for every $1 \leq i \leq m$, $\sigma(y_i) = v^+(q_i)$. By assumption,$\bigvee_{1 \leq i \leq n}(\neg l_{i1} \wedge \neg l_{i2} \wedge \neg l_{i3})$ holds under v^+, so there is some $1 \leq j \leq n$ such that $\neg l_{j1} \wedge \neg l_{j2} \wedge \neg l_{j3}$ evaluates to true under v^+ and by construction of T, it follows that for all $1 \leq k \leq 3$:

$$\begin{gathered} T \not\models \exists z_2 z_3 (Cl(c_j)/L(l_{jk})/P(z_2)/Oc(0) \\ \wedge \\ V(2)/[Val(z_2) \to^* Val(0)] \\ \wedge \\ Cl(c_j)/L(l_{jk})/P(z_3)/Oc(1) \\ \wedge \\ V(2)/[Val(1) \to^* Val(z_3)] \\ \wedge \\ \bigwedge_{1 \leq k \leq l} Cl(d_j)/L(l_{jk})/Q(q_k)/Oc(\sigma(y_k))) \end{gathered}$$

But this is a contradiction, so $T \not\models q'$. As the size of q and q' is polynomial in the size of φ, this completes the proof of the reduction. □

C. PROOFS FROM SECTION 5

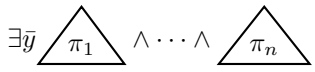
In this appendix we provide a proof of Theorem 5.4. Without loss of generality we proceed as in the proof of Theorem 3.1 and restrict to Boolean queries. The idea behind the proof of the NP upper bound is as follows. Given two queries $q, q' \in CQ(\Downarrow, \rightarrow)$, we show that $q \subseteq q'$ if and only if there is a homomorphism $h : t_{q'} \rightarrow (t_q)^*$. To prove this, we first construct out of t_q some "canonical" complete tree T such that there is a natural one to one homomorphism $h_1 : (t_q)^* \rightarrow T$. Assuming $q \subseteq q'$, we then infer that there exists another homomorphism $h_2 : t_{q'} \rightarrow T$. We finally construct the homomorphism h from h_1 and h_2 by letting $h(x) = h_1^{-1}(h_2(x))$.

DEFINITION C.1 (RIGIDIFICATION OF A CONJUNCTIVE QUERY). *Let $q \in CQ(\Downarrow, \rightarrow)$ and let $\heartsuit \in \mathcal{L}$ be a label and $\sharp \in \mathcal{D}$ a data value which both have no occurrence in q. We define q^{rig} by induction as follows:*
For $q := \exists \bar{y} \bigwedge_{i=1}^{n} \pi_i(\bar{z}_i)$ we define $q^{rig} := \exists \bar{y}\, \heartsuit(\sharp)/[(\pi_1(\bar{z}_1))^R \rightarrow \heartsuit(\sharp) \rightarrow \ldots \rightarrow \heartsuit(\sharp) \rightarrow (\pi_n(\bar{z}_n))^{rig}]$;
For $\pi(\bar{z}) := (\alpha(x)[\mu_1, \ldots, \mu_k]//[\mu'_1, \ldots, \mu'_l])^{rig}$, we define $\pi(\bar{z})^{rig}$ as

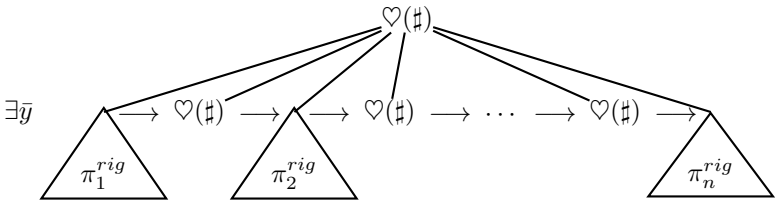
$$\alpha(x)[(\mu_1)^{rig} \rightarrow \heartsuit(\sharp) \rightarrow \ldots \rightarrow \heartsuit(\sharp) \rightarrow (\mu_k)^{rig} \rightarrow \heartsuit(\sharp)[(\mu'_1)^{rig}] \rightarrow \ldots \rightarrow \heartsuit(\sharp)[(\mu'_l)^{rig}]];$$

Finally for $\mu := \pi_1 \rightarrow \ldots \rightarrow \pi_m$ we define $\mu^{rig} := (\pi_1)^{rig} \rightarrow \ldots \rightarrow (\pi_m)^{rig}$.

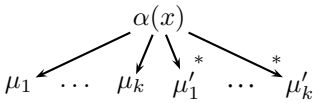
The following drawings may help the reader to understand the definition of q^{rig}.



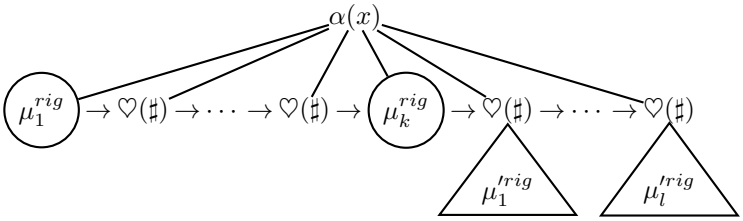
The query q



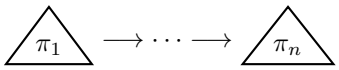
The query q^{rig}



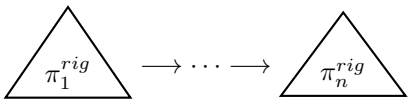
A pattern $\pi(\bar{z})$



The tree $\pi(\bar{z})^R$



An ordered forest of patterns $\mu(\bar{z})$

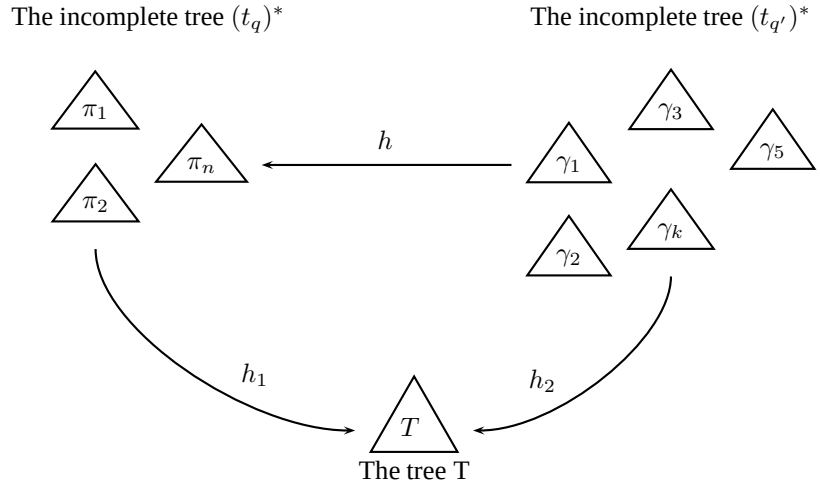


The tree $\mu(\bar{z})^{rig}$

DEFINITION C.2. *Let $q \in CQ(\Downarrow, \rightarrow)$. There is a "natural" homomorphism $h^{rig} : t_q \rightarrow (t_q)^{rig}$ that can be defined as follows. First construct $q_{\mathbb{N}}$ out of q by replacing every distinct occurrence of every $L \in \mathcal{L}$ by a fresh label L_n, where $n \in \mathbb{N}$. There is only one homomorphism $h^{rig} : t_{q_{\mathbb{N}}} \rightarrow t_{q^{rig}_{\mathbb{N}}}$ and it maps each node in $t_{q_{\mathbb{N}}}$ to the only node in $t_{q^R_{\mathbb{N}}}$ which carries the same label. Now if we remove indexes on labeling symbols, the mapping $h^{rig} : t_q \rightarrow t_{q^{rig}}$ remains a homomorphism.*

DEFINITION C.3. *Let t be an incomplete tree. We define t^* inductively out of t by adding in $\downarrow^*$ every pair of nodes which is in the reflexive transitive closure of $\downarrow \cup \downarrow^*$ in t.*

PROPOSITION C.4. *Let $q \in CQ(\Downarrow, \rightarrow)$ and let T be a data tree obtained from $t_{q^{rig}}$ by uniformly substituting fresh distinct constants for distinct nulls using a mapping f. Then $f \circ h^{rig} : (t_q)^* \rightarrow T$ is a one-to-one homomorphism and for all $w, w' \in (t_q)^*$, for all $R \in \{\downarrow, \downarrow^*, \rightarrow\}$, wRw' if and only if $f \circ h^{rig}(w) R f \circ h^{rig}(w')$.*



THEOREM C.5. *The problem $CQ_{\subseteq}(\Downarrow, \rightarrow)$ is in* NP.

PROOF. Let $q, q' \in \mathrm{CQ}(\Downarrow, \rightarrow)$, we show that the following are equivalent:

1. there is a homomorphism $h : (t_{q'})^* \rightarrow (t_q)^*$;
2. $q \subseteq q'$, i.e., for every data tree T, if there is a homomorphism $h_1 : t_q \rightarrow T$, then there is a homomorphism $h_2 : t_{q'} \rightarrow T$.

As 1. can be tested in NP, the result follows.

$1 \rightarrow 2$ Assume a homomorphism $h : (t_{q'})^* \rightarrow (t_q)^*$. Now let $T \models q$, so there exists a homomorphism $h_1 : (t_q)^* \rightarrow T$. We want to show $T \models q'$, so we need to find a homomorphism $h_2 : t_{q'} \rightarrow T$. Homomorphisms being preserved by composition, by transitivity we can simply put $h_2(x) = h_1(h(x))$.

$2 \rightarrow 1$ Let $q, q' \in \mathrm{CQ}_{\mathrm{XML}}(\downarrow, \Downarrow, \rightarrow)$ and assume $q \subseteq q'$, i.e., for every data tree T, $T \models q$ implies $T \models q'$. We assume without loss of generality that the label $\heartsuit$ and the data value $\sharp$ do not occur neither in q nor in q' and we first construct $(t_{q^{rig}})^*$ out of q. By uniformly substituting nulls with fresh distinct arbitrary data values from $\mathcal{C}$ using some suitable mapping f, we now obtain a structure which is isomorphic to some data tree T such that, by Proposition C.4, $T \models q$ and there exists an injective homomorphism $h_1 : (t_q)^* \rightarrow T$, where $h_1 = f \circ h^{rig}$. Now by 2., there exists a homomorphism $h_2 : t_{q'} \rightarrow T$ and it follows from the definition of $(t_{q'})^*$ that $h_2 : (t_{q'})^* \rightarrow T$ is also a homomorphism. Let D be the domain of T. As $Im(h_1) = \{n \in D \mid \lambda(n) \neq \heartsuit\} \cup \{c \in C \mid n \in D, \lambda(n) \neq \heartsuit, \rho(n) = c\}$ and $\heartsuit$ and $\sharp$ occur nowhere in q', it follows that $Im(h_2) \subseteq Im(h_1)$. The homomorphism h_1 being one-to-one, we can now construct a mapping $h : (t_{q'})^* \rightarrow (t_q)^*$, where $h(x)$ is the unique element in $h_1^{-1}(h_2(x))$ (such a set is indeed always a singleton that we identify here with its unique element). Now we claim that h is a homomorphism, as it satisfies the following properties:

(a) If wRw' in $(t_{q'})^*$, then $h(w)Rh(w')$ in $(t_q)^*$, when R is one of $\downarrow, \rightarrow, \Downarrow$.
Let wRw' in $(t_{q'})^*$ for some $R \in \{\downarrow, \downarrow^*, \rightarrow\}$. As h_2 is a homomorphism, it follows that $h_2(w)Rh_2(w') \in T$. As $\heartsuit$ does not occur in $(t_{q'})^*$, we know that $\lambda(h_2(w)) \neq \heartsuit$ and $\lambda(h_2(w')) \neq \heartsuit$. But this implies that both $h_2(w)$ and $h_2(w')$ have a pre-image by h_1 in $(t_q)^*$ and by Proposition C.4 also $h_1^{-1}(h_2(w))Rh_1^{-1}(h_2(w')) \in (t_q)^*$.

(b) If $\lambda(w) = L$ in $(t_{q'})^*$, then $\lambda(h(w)) = L$ in $(t_q)^*$.
As h_2 is a homomorphism, $\lambda(h_2(w)) = L$ in T and as h_1 is a homomorphism, $\lambda(h^{-1}(h_2(w))) = L$.

(c) $h(c) = c$ for all $c \in \mathcal{D}$.
Let c be a constant occurring in q'. As h_2 is a homomorphism, $h_2(c) = c$. By construction of T, constants occurring in T but not in q are fresh, i.e., they occur neither in q nor in q'. So c occurs also in q. As h_1 is a homomorphism, it follows that $h_1^{-1}(h_2(c)) = c$.

(d) $h(\rho(w)) = \rho(h(w))$ for each $w \in t_{q'}$.
As h_1 is a onto homomorphism mapping distinct nulls to distinct fresh constants, for all $w \in Im(h_1)$, $h_1^{-1}(\rho(w)) = \rho(h_1^{-1}(w))$. As $Im(h_2) \subseteq Im(h_1)$, it follows that $h_1^{-1}(\rho(h_2(w))) = \rho(h_1^{-1}(h_2(w)))$.

□

D. PROOFS FROM SECTION 6

In the following, we refer for patterns in $\mathbf{\Pi}(\downarrow, \rightarrow)$ as "rigid patterns".

PROPOSITION D.1. *Let π be a rigid pattern, we define $\pi^{\text{–}}$ recursively as the disjunction of all rigid patterns extending π with one single node labeled with wildcard and with a fresh variable over data values. The size of $\pi^{\text{–}}$ is polynomial in the size of π.*

PROOF. Let n be the number of nodes (or sub-patterns) in π. There are no more than $2n + 1$ways of adding one single node to a rigid pattern. The operation can indeed only be performed as follows:

- the new node becomes the parent of the root of π, i.e., out of π we form $\_(x)/\pi$;
- the new node becomes a child of a node in π, if π has no children, then it becomes its only child, otherwise it can only become either its first child or its last child, i.e., for a sub-pattern $a(y)[\pi_1 \rightarrow \ldots \rightarrow \pi_n]$ in π, we can form either only $a(y)/\_(x)$ if the sequence of the π_i's is empty, or $a(y)[\_(x) \rightarrow \pi_1 \rightarrow \ldots \rightarrow \pi_n]$ and $a(y)[\pi_1 \rightarrow \ldots \rightarrow \pi_n \rightarrow \_(x)]$ otherwise.

□

EXAMPLE D.2. *Consider the rigid pattern $l(a)/l(b)$. We can construct $(l(a)/l(b))^{\text{–}}$ as the following union of patterns:*

$$\_(x)/l(a)/l(b) \ \cup \ l(a)/[l(b) \rightarrow \_(x)] \ \cup \ l(a)/[\_(x) \rightarrow l(b)] \ \cup \ l(a)/l(b)/\_(x)$$

THEOREM D.3. *The problem $\mathrm{UCQ}_{\subseteq}(\downarrow, \rightarrow, \_)$ is Π_2^p-hard.*

PROOF. We proceed by reduction from $\forall\exists 3CNF$. We only need to adapt the proof of Theorem 4.2. Given as input a formula

$$\varphi := \forall p_1 \ldots \forall p_l \exists q_1 \ldots \exists q_m \bigwedge_{1 \leq i \leq n} (l_{i1} \vee l_{i2} \vee l_{i3}),$$

we construct queries $q_{rigid}, q'_{rigid} \in \mathrm{UCQ}(\downarrow, \rightarrow, \_)$ such that:

$$\varphi \quad \text{is true} \qquad \text{if and only if} \qquad q_{rigid} \subseteq q'_{rigid}.$$

We construct q_{rigid}, q'_{rigid} using the same alphabet of node labels as in the proof of Theorem 4.2 as follows:

$$q_{rigid} = \exists x_1 \ldots \exists x_l (\bigwedge_{1 \leq i \leq l} Val(x_i) \wedge \Pi)$$

$$q'_{rigid} = \exists x \exists x_1 \ldots \exists x_l (\Pi)^{\text{–}}$$

$$\vee$$

$$\exists y_1 \ldots \exists y_m (\bigwedge_{1 \leq i \leq m} Val(y_i)$$

$$\wedge$$

$$\bigwedge_{1\leq j\leq n} \exists z(Cl(c_j)/L(z)/P(0)/Oc(0)$$

$$\wedge$$

$$Cl(c_j)/L(z)/P(1)/Oc(1)$$

$$\wedge$$

$$\bigwedge_{1\leq k\leq m} Cl(d_j)/L(z_1)/Q(q_k)/Oc(y_k))$$

where Π is a rigid pattern which root is a node labeled $Val(0)$ and which has as first child a single node pattern labeled $Val(1)$. The rest of its children is constituted by the ordered sequence of the rigid Π_{ij}'s patterns defined in the proof of Theorem 4.2.

The intuition behind the reduction is the following. Let T be a data tree satisfying q_{rigid} and not satisfying $\exists x \exists x_1 \ldots \exists x_l(\Pi)$-. This means that there is a homomorphism from $(t_{q_{rigid}})$ to T which maps all the l nulls in $(t_{q_{rigid}})$ to Boolean values. The set of all trees satisfying this property will thus allow to encode the set of all possible valuations of the p_i's. □

THEOREM D.4. *The problem* $\mathrm{UCQ}_{\subseteq}(\Downarrow, \_^{\neg r})$ *is* Π_2^p*-hard.*

PROOF. We proceed by reduction from $\forall\exists 3CNF$ and just adapt the proof of Theorem 4.2. Given as input a formula

$$\varphi := \forall p_1 \ldots \forall p_l \exists q_1 \ldots \exists q_m \bigwedge_{1\leq i\leq n} (l_{i1} \vee l_{i2} \vee l_{i3}),$$

we construct queries $q_{\Downarrow}, q'_{\Downarrow} \in \mathrm{UCQ}(\Downarrow, \_^{\neg r})$ such that:

$$\varphi \quad \text{is true} \qquad \text{if and only if} \qquad q_{\Downarrow} \subseteq q'_{\Downarrow}.$$

We construct $q_{\Downarrow}, q'_{\Downarrow}$ using the same alphabet of node labels as in the proof of Theorem 4.2 as follows:

$$q_{\Downarrow} = \exists x_1 \ldots x_l(\bigwedge_{1\leq i\leq n, 1\leq j\leq 3} \Pi_{ij} \wedge \Pi'_{Val})$$

$$q'_{\Downarrow} = \exists x_1 \ldots \exists x_{2l+1} Val(0)/\_(x_1)/\ldots/\_(x_{2l+1})$$

$$\vee$$

$$\exists y_1 \ldots \exists y_m(\bigwedge_{1\leq i\leq m} Val(y_i)$$

$$\wedge$$

$$\bigwedge_{1\leq j\leq n} \exists z(Cl(c_j)/L(z)/P(0)/Oc(0)$$

$$\wedge$$

$$Cl(c_j)/L(z)/P(1)/Oc(1)$$

$$\wedge$$

$$\bigwedge_{1\leq k\leq m} Cl(d_j)/L(z_1)/Q(q_k)/Oc(y_k))$$

where Π'_{Val} is a pattern of the following form:

$$Val(0)//Val(x_1)//Val(1)/Val(0)//Val(x_2)//Val(1)/\ldots/Val(0)//Val(x_{l-1})//Val(1)/Val(0)//Val(x_l)//Val(1).$$

The intuition behind the reduction is the following. Let T be a data tree satisfying $q_{\Downarrow}$ and not satisfying $\exists x_1 \ldots \exists x_{2l+1} Val(0)/\_(x_1)/\ldots/\_(x_{2l+1})$. This means that there is a homomorphism from $(t_{q_{\Downarrow}})$ to T which maps all the l nulls in $(t_{q_{\Downarrow}})$ to Boolean values. The set of all trees satisfying this property will thus allow to encode the set of all possible valuations of the p_i's. □

E. PROOFS FROM SECTION 7

In this section we assume Σ to be the finite alphabet of the schema.

We first prove the 2EXPTIME upper bound (Theorem 7.1). It follows from a translation to a sequence of intersection emptiness problems of non-deterministic unranked tree automata. Before giving the proof, we give a few definitions and preliminary results.

For a query $q \in \text{CQ}(\Downarrow, \Rightarrow, \_)$, we define the *associated incomplete data tree* t_q similarly as in Section 5. The difference is that t_q is now also allowed to have nodes labeled with the wildcard "_". A homomorphism from t_q to a data tree t is now also allowed to send a node labeled _ to a node labeled with some $a \in \mathcal{L}$.

Analogously as in Section 5, we have the following proposition:

PROPOSITION E.1. *Let t be a data tree, and $q(\bar{x})$ is a query from CQ$(\Downarrow, \Rightarrow, _)$. Then $t \models q(\bar{v})$ iff there is a homomorphism $h : t_q \to t$ so that $h(\bar{x}) = \bar{v}$.*

The proof for the upper bound reduces the problem from data trees to (non-data) unranked trees with respect to a certain query q. In particular, for a finite alphabet Σ and a query q of size $n = |q|$, we define a finite alphabet Σ_q as follows. Denote by $s_1 \cdots s_n$ the nodes in t_q, the alphabet Σ_q is of the form $\Sigma \times \{d_1, \ldots, d_n, *\}$ such that if a node s_i is associated to a constant $c \in \mathcal{D}$ in the query q, then the letter $d_i = c$. We denote the set of (non-data) unranked trees over alphabet Σ_q by $\mathcal{T}_q$.

We define the function $f_{\neq}$ that maps trees from $\mathcal{T}_q$ to data trees. Given a tree $t = \langle D, \downarrow, \rightarrow, \lambda\rangle$ in $\mathcal{T}_q$, then $f_{\neq}(t)$ is the data tree $\langle D, \downarrow, \rightarrow, \lambda', \rho\rangle$ which is obtained from t as follows

1. If $\lambda(v) = (a, d_i)$, for $a \in \Sigma$ and $i \in \{1, \ldots, n\}$, then $\lambda'(v) = a$ and $\rho(v) = d_i$, that is, node v of $f_{\neq}(t)$ has label a and data value d_i.
2. If $\lambda(v) = (a, *)$, for $a \in \Sigma$, then $\lambda'(v) = a$ and $\rho(v)$ is a new data value, that is, node v of $f_{\neq}(t)$ has label a and a data value that does not appear anywhere else in $f_{\neq}(t)$.

We now prove the following lemma:

LEMMA E.2. *Let q be a query in CQ$(\Downarrow, \Rightarrow, _)$. Then one can construct in exponential time an NTA $\mathcal{A}_{\text{fin}}$ over alphabet Σ_q such that $t \in \mathcal{L}(\mathcal{A}_{\text{fin}})$ if and only if $f_{\neq}(t) \models q$ for each Σ_q-tree t.*

PROOF. The proof is similar to a proof in [10]. Let $t_q = (N, V, \downarrow, \downarrow^*, \rightarrow, \rightarrow^*, \lambda, \rho)$ be the incomplete data tree associated with q. Let n be the number of nodes in t_q. Our aim is to construct the NTA $\mathcal{A}_{\text{fin}}$ that works as follows.

When reading a Σ_q-tree t, the automaton $\mathcal{A}_{\text{fin}}$ guesses a homomorphism $h : t_q \to t$ that witnesses $f_{\neq}(t) \models q$ (cf Proposition E.1), if it exists. More precisely, $\mathcal{A}_{\text{fin}}$ guesses the nodes of t which are the homomorphic images of nodes of t_q and checks whether the correct relations hold between the guessed nodes.

Intuitively, a state of $\mathcal{A}_{\text{fin}}$ is of the form (X_a, X_h, X_d, D), where $D : V \to \{d_1, \ldots, d_n, *\}$ is a total function and X_a, X_h, and X_d are subsets of N such that

- X_a is the set of nodes in N that $\mathcal{A}_{\text{fin}}$ guesses to be mapped to the *ancestors of the current node*,
- X_h is the set of nodes in N that $\mathcal{A}_{\text{fin}}$ guesses to be mapped *on the current node*, and

- X_d is the set of nodes in N that $\mathcal{A}_{\text{fin}}$ guesses to be mapped on *descendants of the current node*.

Since $\mathcal{A}_{\text{fin}}$ guesses a (well-defined) homomorphism, we only guess one target node for every node in N. Hence, for each state (X_a, X_h, X_d, D), the pairwise intersections of X_a, X_h, and X_d are empty.

In order to define $\mathcal{A}_{\text{fin}} = (Q_A, \Sigma_q, \delta_A, F_A)$ formally, we specify Q_A, F_A, and δ_A:

Q_A**:** The state set Q_A of $\mathcal{A}_{\text{fin}}$ is the maximal subset of $2^N \times 2^N \times 2^N \times \{d_1, \ldots, d_n, *\}^V$ such that the following conditions hold: For each $(X_a, X_h, X_d, D) \in Q_A$,

(Q1) the pairwise intersections of X_a, X_h, and X_d are empty,

(Q2) for each $x, y \in X_h$, the labels of x and y are the same or one of them is a wildcard; that is, either $\lambda(x) = \lambda(y)$ or $\lambda(x) = \_$ or $\lambda(y) = \_$,

(Q3) for each $x \in X_h$ and each $y \in N$ such that $y \downarrow x$ in t_q, we have that $y \in X_a$, and

(Q4) for each $x \in X_h$ and each $y \in N$ such that $y \downarrow^* x$ in t_q, we have that $y \in X_h \cup X_a$.

F_A**:** A state (X_a, X_h, X_d, D) of $\mathcal{A}_{\text{fin}}$ is in F_A if and only if

(F1) X_a is empty; and

(F2) X_h and X_d partition N, i.e., $N = X_h \uplus X_d$.

δ_A**:** contains all transitions of the form

$$\delta_A\big((X_a, X_h, X_d, D), (a, \Delta)\big) = L, \tag{$\dagger$}$$

where

(D1) for each $x \in X_h$, $D(x) = \Delta$;

(D2) for each $(X_a^1, X_h^1, X_d^1, D^1) \cdots (X_a^m, X_h^m, X_d^m, D^m) \in L$, we have $D^1 = \cdots = D^m = D$; and

(Hor) for every string $(X_a^1, X_h^1, X_d^1, D^1) \cdots (X_a^m, X_h^m, X_d^m, D^m) \in L$, the following holds:

(a) $X_d = X_h^1 \uplus \cdots \uplus X_h^m \uplus X_d^1 \uplus \cdots \uplus X_d^m$;

(b) if $x \in X_h$ and $x \downarrow y$ in t_q then there is an $i = 1, \ldots, m$ with $y \in X_h^i$;

(c) for each $i = 1, \ldots, m$, $X_a^i = X_a \cup X_h$; and

(d) for each $i = 1, \ldots, m$, if $x \in X_h^i$ and

- if $x \to y$ in t_q, then $i < m$ and $y \in X_h^{i+1}$;
- if $x \to^* y$ in t_q, then there exists a j, $i \leq j \leq m$ such that $y \in X_h^j$.

In order to complete the proof of the lemma, we need to prove that

(1) $\mathcal{A}_{\text{fin}}$ can be constructed from q in exponential time; and

(2) $\mathcal{L}(\mathcal{A}_{\text{fin}}) = \{t \in \mathcal{T}_q \mid f_{\neq}(t) \models q\}$

(1): It is clear that Q_A and F_A can be computed in time exponential in $|q|$. For δ_A, we prove that we can compute an non-deterministic finite string automaton (NFA) $\mathcal{N}$ that accepts, for every $(X_a, X_h, X_d, D) \in Q_A$ and $(a, \Delta) \in \Sigma_q$, the language L in the rule

$$\big((X_a, X_h, X_d, D), (a, \Delta)\big) \to L.$$

As $\mathcal{N}$ only reads symbols from Q_A, we don't need to check anymore that (Q1)–(Q4) hold. Furthermore, (D1) and (D2) also do not need to be checked by $\mathcal{N}$. These conditions need to be checked by the algorithm that constructs A, when deciding whether or not to define a transition rule of the form ($\dagger$). Hence, we only have to enforce (Hor.a)–(Hor.d).

We next describe $\mathcal{N}$'s accepting condition and the information that $\mathcal{N}$ needs to remember when reading a string. Since $\mathcal{N}$ only needs to maintain a polynomial amount of information when reading a string, it should be clear that $\mathcal{N}$ needs only an exponentially large set of states. A state of $\mathcal{N}$ consists of $(X_h^{\cup}, X_d^{\cup}, Y_{ns}, Y_{ns*})$, where the components are defined as follows. When reading a prefix $(X_a^1, X_h^1, X_d^1, D) \cdots (X_a^k, X_h^k, X_d^k, D)$ of $(X_a^1, X_h^1, X_d^1, D) \cdots (X_a^n, X_h^n, X_d^n, D)$,

- $X_h^{\cup} := X_h^1 \cup \cdots \cup X_h^k$,
- $X_d^{\cup} := X_d^1 \cup \cdots \cup X_d^k$,
- $Y_{ns} := \{y \mid x \in X_h^k \text{ and } x \to y \text{ in } t_q\}$,
- $Y_{ns*} := \{y \mid \exists 1 \leq i \leq k, x \in X_h^i, y \notin X_h^i \cup \cdots \cup X_h^k \text{ and } x \to^* y \text{ in } t_q\}$.

When reading symbol $(X_a^{k+1}, X_h^{k+1}, X_d^{k+1}, D)$, N checks whether

- $X_h^{k+1} \cap (X_h^{\cup} \cup X_d^{\cup}) = \emptyset$, to partially ensure (Hor.a);
- $X_d^{k+1} \cap (X_h^{\cup} \cup X_d^{\cup}) = \emptyset$, to partially ensure (Hor.a);
- $X_a^{k+1} = X_a \cup X_h$, to ensure (Hor.c); and
- $Y_{ns} \subseteq X_h^{k+1}$, to ensure (Hor.d)'s NextSibling-constraint.

and it changes its state to $(X'^{\cup}_h, X'^{\cup}_d, Y'_{ns}, Y'_{ns*})$ as follows:

- $X'^{\cup}_h = X_h^{\cup} \cup X_h^{k+1}$;
- $X'^{\cup}_d = X_d^{\cup} \cup X_d^{k+1}$;
- $Y'_{ns} = \{y \mid x \in X_h^{k+1} \text{ and } x \to y \text{ in } t_q\}$;
- $Y'_{ns*} = (Y_{ns*} - X_h^{k+1}) \cup \{y \mid x \in X_h^{k+1}, y \notin X_h^{k+1}, \text{ and } x \to^* y \text{ in } t_q\}$.

Finally, $\mathcal{N}$ accepts if

- $X_d = X_h^{\cup} \cup X_d^{\cup}$, to ensure (R2.a), together with the above conditions on the transitions;
- for each $x \in X_h$ such that $x \downarrow y$ in t_q, $x \in X_h^{\cup}$, to ensure (Hor.b);
- $Y_{ns} = \emptyset$, to ensure (Hor.d)'s $\to$ relations; and
- $Y_{ns*} \subseteq X_h^k$, to ensure (Hor.d)'s $\to^*$ relation.

(2): We show that $L(A) = \{t \in \mathcal{T}_q \mid f_{\neq}(t) \models q\}$. For the inclusion from left to right, take $t \in L(A)$. Consider the homomorphism h from t_q to t induced by an accepting run of A on t by setting $h(x) = v$ if the run assigned state (X_a, X_h, X_d, D) to v with $x \in X_h$. It is easy to show by induction on trees t that the homomorphism h is a homomorphism from t_q to $f_{\neq}(t)$ and therefore witnesses that $q \models f_{\neq}(t)$. Conversely, if h is a homomorphism from t_q to $f_{\neq}(t)$ then the same homomorphism allows us to construct an accepting run of $\mathcal{A}_{\text{fin}}$ on t. □

The transformation $f_{\neq}$ we defined associates a data tree to a Σ_q-tree. We now define a transformation g_q to associate a Σ_q-tree to a data tree satisfying a given query.

More precisely, let q be a CQ$(\Downarrow, \Rightarrow, _)$ and let $t_q = \langle N, V, \downarrow, \downarrow^*, \to, \to^*, \lambda, \rho\rangle$ be its associated incomplete data tree with nodes $N = \{s_1, \ldots, s_n\}$ and let t be a data tree such that $t \models q$. Furthermore, let $h : t_q \to t$ be a homomorphism that witnesses that $t \models q$. Then we write $g_q(t, h)$ for the Σ_q-tree resulting from t as follows.

- If, for some i, $h(u_i) = v$ and $\lambda^T(v) = a$, then v gets label (a, d_j) where j is minimal with $\rho(h(u_j)) = \rho(v)$.
- Otherwise, v gets label $(a, *)$ if $\lambda^T(v) = a$.

Intuitively, the tree $g_q(t, h)$ is obtained by relabeling the data in the tree t using letters from $\{d_1, \ldots, d_n\}$ for the nodes which are in the image of the homomorphism h and using the letter $*$ for all the other nodes.

The following lemma is analogous to a lemma from [10].

LEMMA E.3. *Let $q, q' \in$ CQ$(\Downarrow, \Rightarrow, _)$ where t_q has n nodes and let t be a data tree such that $t \models q$ but $t \not\models q'$. Then the following hold:*

(a) $f_{\neq}(g_q(t, h)) \models q$ *and*

(b) $f_{\neq}(g_q(t, h)) \not\models q'$.

PROOF. By definition of g_q and $f_{\neq}$, it is straightforward that the same homomorphism h witnesses that $f_{\neq}(g_q(t, h)) \models q$. This proves (a).
We now prove (b). Towards a contradiction, assume $f_{\neq}(g_q(t, h)) \models q'$. Let h' be a homomorphism from $t_{q'}$ to $f_{\neq}(g_q(t, h))$. By definition of $f_{\neq}(g_q(t, h))$, the only difference between t and $f_{\neq}(g_q(t, h))$ is the data values. Furthermore, for every pair of nodes s_1, s_2 in $f_{\neq}(g_q(t, h))$, if $\rho(s_1) = \rho(s_2)$ in $f_{\neq}(g_q(t, h))$, then we also have that $\rho(s_1) = \rho(s_2)$ in t. This implies that h' would also be a homomorphism from t'_q to t which contradicts that $t \not\models q'$. □

We can now prove the following upper bound:

Theorem 7.1. *The problem* $\text{BCCQ}_{\subseteq}(\Downarrow, \Rightarrow, \_)$ *under schemas can be solved in* 2EXPTIME.

PROOF. To prove this upper bound, we show that satisfiability of $\mathrm{BCCQ}(\Downarrow, \Rightarrow, \_)$ queries can be solved in 2EXPTIME, from which the overall upper bound follows. Let q be a query from $\mathrm{BCCQ}(\Downarrow, \Rightarrow, \_)$ that contains the CQs $q_1, \ldots, q_k$.

The query q is satisfiable if and only if there exists a valuation $\chi : \{1, \ldots, k\} \to \{\text{true}, \text{false}\}$ and a tree t such that,

- the propositional formula obtained from q by replacing each q_i with $\chi(q_i)$ is true;

and for every $i = 1, \ldots, k$,

- $t \models q_i$ if $\chi(i) = \text{true}$ and $t \not\models q_i$ if $\chi(i) = \text{false}$.

We say that t models q under χ.

The 2EXPTIME algorithm will iterate through all possible valuations χ and test whether there exists a tree t that models q under χ. Since there are only exponentially many valuations, it is sufficient to show that we can perform the test for a single, fixed, valuation in 2EXPTIME.

Therefore, fix a valuation χ and let P be the conjunction of all q_i such that $\chi(i) = \text{true}$. Furthermore, let $\{q'_1, \ldots, q'_m\}$ be the set of all q_i such that $\chi(i) = \text{false}$. Let n be the size of of t_P. According to Lemma E.3, there exists a tree T_χ that models q under χ if and only if there exists a homomorphism h and a tree of the form $f_{\neq}(g_P(T_\chi, h))$ that models q under χ. The latter means that $f_{\neq}(g_P(T_\chi, h)) \models P$ and $f_{\neq}(g_P(T_\chi, h)) \not\models q'_i$ for every $i = 1, \ldots, m$.

According to Lemma E.2, we can build, for the query P, an NTA $\mathcal{A}^P_{\text{fin}}$ such that $f_{\neq}(g_P(T_\chi, h)) \models P$ if and only if $g_P(T_\chi, h) \in L(\mathcal{A}^P_{\text{fin}})$. Similarly for every query q'_i, we can build an NTAs $\mathcal{A}^{q_i}_{\text{fin}}$ such that $f_{\neq}(g_P(T_\chi, h)) \models q_i$ if and only if $g_P(T_\chi, h) \in L(\mathcal{A}^{q_i}_{\text{fin}})$.

The latter implies that it is equivalent to test whether there exists a tree t that models q under χ and to test whether $L(\mathcal{A}^P_{\text{fin}}) \cap \overline{L(\mathcal{A}^{q'_1}_{\text{fin}})} \cap \cdots \cap \overline{L(\mathcal{A}^{q'_m}_{\text{fin}})}$ is non-empty.

We still need to discuss how schema information in the form of a tree automaton can be incorporated. But that is very easy in the present approach: If $\mathcal{A}$ is the tree automaton (over Σ) under which we want to test whether q is satisfiable, then let $\mathcal{A}_q$ be the tree automaton (over Σ_q) that accepts a Σ_q-tree if and only if its projection on the corresponding Σ-tree is accepted by $\mathcal{A}$. (More formally, for every label of the form (a, x), the automaton $\mathcal{A}_q$ simply ignores x and simulates $\mathcal{A}$.)

As such, we only need to test if $L(\mathcal{A}_q) \cap L(\mathcal{A}^P_{\text{fin}}) \cap \overline{L(\mathcal{A}^{q'_1}_{\text{fin}})} \cap \cdots \cap \overline{L(\mathcal{A}^{q'_m}_{\text{fin}})}$ is non-empty. The latter test can be performed in 2EXPTIME by standard techniques on automata. □

Theorem 7.2. *Containment of CQs under schemas is* 2EXPTIME*-hard; i.e., the problem* $\mathrm{CQ}_\subseteq(\Downarrow, \Rightarrow)$ *under schemas is* 2EXPTIME*-complete.*

The upper bound is immediate from Theorem 7.1.

For the lower bound, we first prove that $\mathrm{UCQ}_\subseteq(\Downarrow, \Rightarrow)$ w.r.t. schema constraints is 2EXPTIME-hard. Then we show how to transform the proof to obtain the lower bound for $\mathrm{CQ}_\subseteq(\Downarrow, \Rightarrow)$.

PROPOSITION E.4. *The problem* $\mathrm{UCQ}_\subseteq(\Downarrow, \Rightarrow)$ *w.r.t. schema constraints is* 2EXPTIME*-hard.*

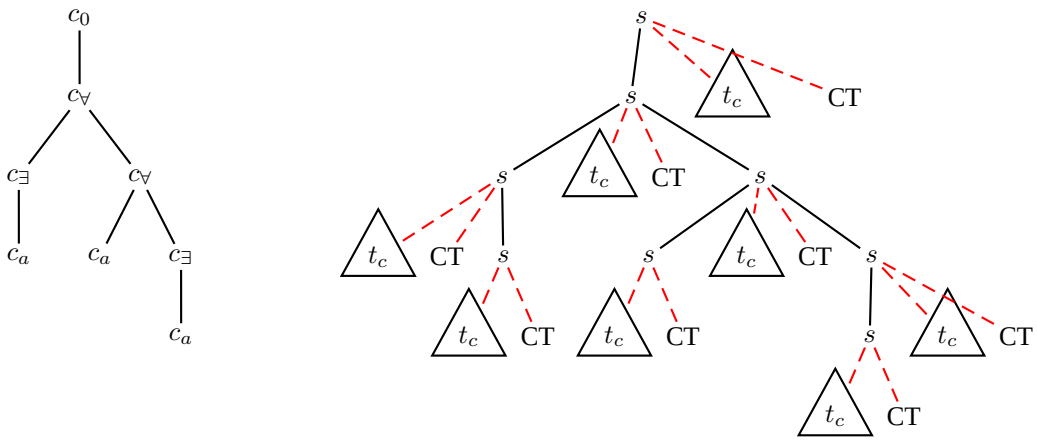
PROOF. We show the lower bound by a reduction from the acceptance problem for alternating exponential space bounded Turing machines. We give an encoding of accepting runs of such machines into data trees. Given such a machine and a word w of length n, we explain how to build a regular language $\mathcal{R}$ and a finite union φ of conjunctive queries such that any data tree satisfying the constraints given by $\mathcal{R}$ and the negation of φ is the encoding of an accepting run of the machine $\mathcal{M}$ on the word w.

The machine $\mathcal{M}$ has no accepting run on the word w iff $q_\emptyset \subseteq_{\mathcal{R}} \varphi$ where $q_\emptyset$ is the trivial query which is always true. Therefore, our UCQ φ will look for all possible *errors* that can happen in runs.

In the following, we define precisely the encoding of accepting runs into a data trees and explain how to build the language $\mathcal{R}$ and the query φ.

Notice that the encoding of the machine has similarities with the one used in proof of Theorem 6 in [10]. However, the present encoding is more complicated due to the fact that our queries are tree-shaped and do not use wildcard.

An alternating Turing machine is a Turing machine $\mathcal{M} = (Q, \Gamma, \Delta, q_0)$ where Γ is the alphabet of the tape containing a special blank symbol $\natural$, the transition relation Δ is a subset of $Q \times \Gamma \times \Gamma \times \{left, right, stay\} \times Q$, the initial state q_0 is



(a) A computation tree. (b) The structure of the corresponding encoding.

Figure 1: The structure of an encoding of a run.

from Q, and the finite set of states Q is partitioned into a set of universal states $Q_\forall$, a set of existential states $Q_\exists$, an accepting state $\{q_a\}$, and a rejecting state $\{q_r\}$. A configuration of $\mathcal{M}$ is a triple $(w, i, q) \in \Gamma^* \times \mathbb{N} \times Q$ where $i \leq |w|$. When $q \in Q_\forall$, we say that the configuration is universal (similarly for existential, accepting, and rejecting). We define successor configurations in the usual way. A computation tree for such a machine on an input word $w \in \Gamma^*$ consists of a tree labeled with configurations such that (0) the root is labeled by the initial configuration (that is, the initial state is q_0, the reading head is at the first position, and the tape contains the word w followed by some blank symbols $\natural$) (1) every node labelled with an existential configuration has exactly one child labeled with a successor (2) every node labelled with a universal configuration c has a child labelled with configuration c' for each successor configuration c' of c, and (3) all leaves are labelled with accepting or rejecting configurations. A computation tree is accepting if and only if it is finite and all leaves labelled with accepting configurations.

Let $\mathcal{M}$ be an alternating exponential space bounded Turing machine. Without loss of generality, we can assume that given any word w of size n the machine never uses more than $2^n - 1$ tape cells.

The encoding of a computation tree.

Let us now define the finite alphabet $\Sigma = \{s, CT, p, 0, 1, q_0\} \cup \Delta \cup \Gamma$.

We encode a computation tree T of the machine into a data tree labeled by Σ as follows:

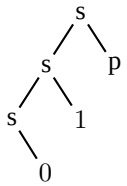
The structure of the tree is described in Figure 1. The tree follows the structure of the computation tree where each corresponding node is labeled with s. To each of these nodes is attached both a node labeled CT and a tree t_c encoding the corresponding configuration and the transition that yields to it (the dashed edges in the figure). We note that, although several subtrees in Figure 1 are labelled t_c, the different occurrences of t_c can actually be differently structured subtrees since they represent different configurations.

We now discuss how a configuration of the computation tree is encoded into a tree t_c. Let (w, i, q) be a configuration of the computation tree obtained by applying the transition $\delta \in \Delta$ to the parent configuration. Recall that a configuration always has length 2^n. We start the encoding of this configuration with a binary tree of depth n in which each node is labelled by s. Because most of the nodes of the encoding will be labeled s, we refer to these s nodes as the skeleton nodes. To each skeleton node but the root, we attach a small tree t_0 for the left children of a skeleton node and t_1 for the right children of a skeleton node as described in Figure 2. Lastly, we use the skeleton leaves to encode (w, i, q) and δ. To each skeleton leaf we attach one node labeled by a letter from $\Gamma \cup \Delta \cup \{q_0\}$, such that taken from left to right the first i new leaves are labeled by the first i letters from w; the $i + 1^{th}$ leaf corresponding to the position of the reading head, is labeled by the letter δ (or, if it is the initial configuration, simply by q_0); the $2^n - (i + 1)$ last leaves are labeled by the $2^n - (i + 1)$ letters of w.

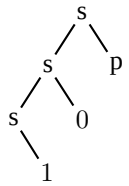
To complete the description of the encoding T we need to describe the data values attached to each node. As we want to use the data to identify the nodes in the whole tree, we ask each node to have a different one.

The language $\mathcal{R}$ and the union of queries φ_{nokey} and $\varphi_{\mathcal{M}}$.

We now need to build a regular tree language $\mathcal{R}$ using the finite alphabet Σ and a union of queries φ such that a data tree



(a) The tree t_0 attached to each left child in t_c.



(b) The tree t_1 attached to each right child in t_c.

Figure 2: The subtrees to distinguish left child from right child in t_c.

$\exists x \;\; a(x)/c_1//b(x)$	$\exists x \;\; c_0/[c_1/c_2//a(x) \to c_5 \to^* c_3/c_4//b(x)]$
$\exists x$ $a(x)$ / c_1 / $*$ / $b(x)$	$\exists x$ c_0; $c_1 \to c_2 \to^* c_3$; $c_1/*/a(x)$; $c_3/*/b(x)$

Figure 3: The queries composing φ_{nokey} (where $a, b, c_0, \ldots, c_2$ represent letters from Σ)

satisfies the constraints given by $\mathcal{R}$ and the negation of φ iff it is an encoding of an accepting run of the machine.

The language $\mathcal{R}$ ensures the general structure of the tree, the labeling of the nodes. It also ensures that the sequence of transitions respects the machine's rules in terms of succession of control states, initial, and final configurations. All these constraints are regular properties of the trees so we can encode them into a (polynomial-size) non deterministic tree automaton.

The data value will allow us to identify the nodes in the whole tree, we will construct the UCQ φ such that each node has a different one.

The union of queries $\varphi = \varphi_{nokey} \cup \varphi_{\mathcal{M}}$ is used to ensure the constraints on data values (φ_{nokey}) and the correct evolution of the content of the tape and the reading head position between consecutive configurations ($\varphi_{\mathcal{M}}$). For each query, we give also a graphical representation that may be easier to parse for the reader.

First we show how to construct φ_{nokey}. This query expresses that there are at least two (different) nodes with the same data value. (So, it expresses that the set of data values is not a key.) The query φ_{nokey} is the union of the queries presented in Figure 3. Notice that the number of these small queries is polynomial in the size of the alphabet Σ and each of them is of constant size (at most 8 nodes). By construction, a data tree labeled by Σ doesn't satisfy φ_{nokey} iff all its nodes have different data values.

We now explain how to construct $\varphi_{\mathcal{M}}$. Intuitively $\varphi_{\mathcal{M}}$ describes all the behaviour that violates the proper evolution in between two consecutive configurations, encoded in trees t_c. In particular, $\varphi_{\mathcal{M}}$ will look for errors in the evolution of the content of the tape and the reading head position between consecutive configurations.

The key points to build $\varphi_{\mathcal{M}}$ is to be able to check that (1) two nodes correspond to the same leaf in two consecutive trees t_c (2) three nodes correspond to two consecutive leaves within a tree t_c. Using these two properties, we can build queries that describes bad evolution of triple of consecutive leaves in between two consecutive tree t_c.

Since φ_{nokey} matches all trees in which at least two nodes have the same data value, we can focus for $\varphi_{\mathcal{M}}$ on encodings in which each node has a different data value. For this reason, *we use data values as a node ID*. This part is rather technical and, for readability sake, we give drawings of tree patterns instead of pure formula (which would be very hard to parse for the reader).

In Figure 4 we describe some useful patterns. The conjunction of patterns $\theta^i_{same}(x, y, x', y')$ is such that if it satisfied by data values $(d_x, d_y, d_{x'}, d_{y'})$, the nodes corresponding to d_x and d_y are either both left or right (skeleton) children at level i in

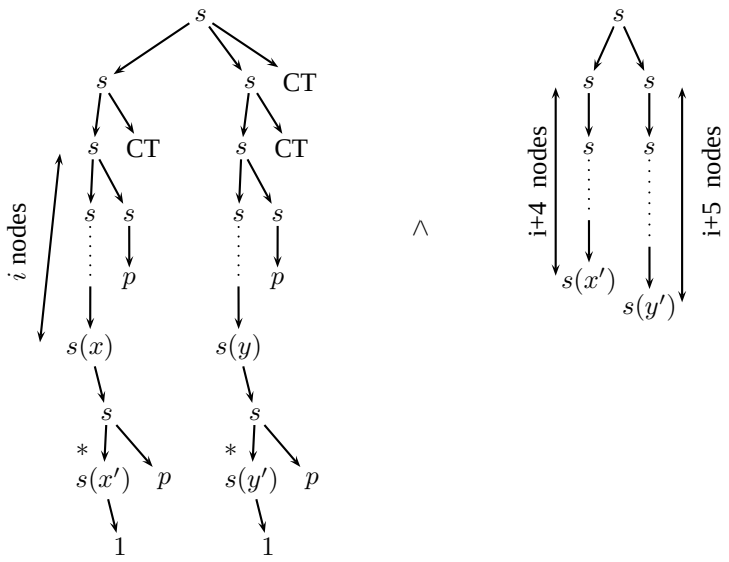


Figure 4: The conjunction of patterns $\theta^i_{same}(x, y, x', y')$.

consecutive t_c. The first pattern ensures that the nodes corresponding to d_x and d_y are skeleton nodes at level i in consecutive t_c; the data values d'_x and d'_y correspond to the respective parents of the 1 node of the subtree of Figure 2 attached to d_x and d_y. The idea is that the two s-nodes d_x and d_y are both left or both right children iff their corresponding 1 nodes have a common ancestor which has distance $i + 5$ from d'_x and $i + 6$ from d'_y. This is ensured by the second pattern of the conjunction.

(1) We can now define the conjunction of patterns $\theta_{same}(x_1, y_1, x'_1, y'_1 \ldots, x_n, y_n, x'_n, y'_n)$ so that, if it is satisfied for some data $(d_{x_i}, d_{y_i}, d_{x'_i}, d_{y'_i})_{i=1..n}$, then the s-nodes corresponding to d_{x_n} and d_{y_n} correspond to the same skeleton leaf of two consecutive t_c trees.

$$\theta_{same}(x_1, y_1, x'_1, y'_1 \ldots, x_n, y_n, x'_n, y'_n) := \bigwedge_{i=1..n} \theta^i_{same}(x_i, y_i, x'_i, y'_i)$$

As x_n and y_n correspond to the nodes we want to characterize, we write for short $\theta_{same}(x_n, y_n, \bar{X})$ instead of $\theta_{same}(x_1, y_1, x'_1, y'_1 \ldots, x_n, y_n, x'_n, y'_n)$.

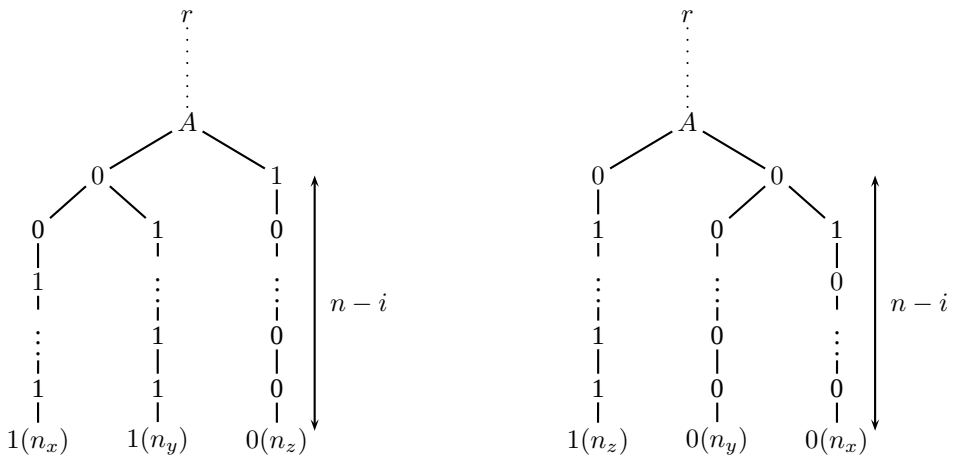
(2) To understand more easily how to identify three consecutive skeleton leaves in a tree t_c let us look at the case of a simple binary tree T_b of depth n rooted in r and where left nodes are labeled with 0 and right ones with a 1. The nodes n_x, n_y, n_z are consecutive leaves in T_b if they are placed as in one of the cases presented in Figure 5 where i represents the depth level of their common ancestor denoted by A in the Figure. Notice that the node A can be labeled either 0 or 1 so altogether we have 4 different possible cases.

We now come back to our encoding. The pattern $\pi^i_{cons1}(x, y, z)$ described in Figure 6 is such that, if it is satisfied for some data (d_x, d_y, d_z), then the corresponding nodes are consecutive leaves of a t_c trees. Moreover, the depth level of their common ancestor in t_c is i and the nodes d_x, d_y and d_z are placed like in Figure 5(a). The small patterns π_0 and π_1 are the direct translation of the trees t_0 and t_1 from Figure 2. This pattern $\pi^i_{cons1}(x, y, z)$ is obtained from Figure 5(a) by changing the $A - node$ into a s-node and by plugging π_0 into nodes 0 and changing this label 0 into s and similarly for 1 nodes.
Using a similar construction, we can define patterns $\pi^i_{cons2}(x, y, z)$ corresponding Figure 5(b).

($\varphi_{\mathcal{M}}$) We now have all the tools to define $\varphi_{\mathcal{M}}$. As we explained before, this union of queries should encode all possible way to violate the proper evolution in between two consecutive trees t_c. We do it looking at triple of consecutive leaves.

The forbidden evolution in two consecutive trees t_c can be summarized as pairs of triples of letters from $\Gamma \cup \Delta \cup \{q_0\}$ representing forbidden evolution of triple of consecutive leaves of t_c trees. Assume (abc, def) is forbidden, we define $\varphi_{abc,def}$ as the union of the following queries for all $i \in \{1..n\}$ and $j \in \{1, 2\}$:

$$\exists x_a, y_b, z_c, x_d, y_e, z_f, \bar{X} \quad \theta_{same}(x_a, x_d, \bar{X}) \wedge \pi^i_{consj}(x_a, y_b, z_c) \wedge \pi^i_{consj}(x_d, y_e, z_f)$$
$$\wedge s(x_a)/a \wedge s(y_b)/b \wedge s(z_c)/c \wedge s(x_d)/d \wedge s(y_e)/e \wedge s(z_f)/f$$



(a) One possible case.

(b) The other possible case.

Figure 5: The nodes n_x, n_y et n_z are consecutive in T_b.

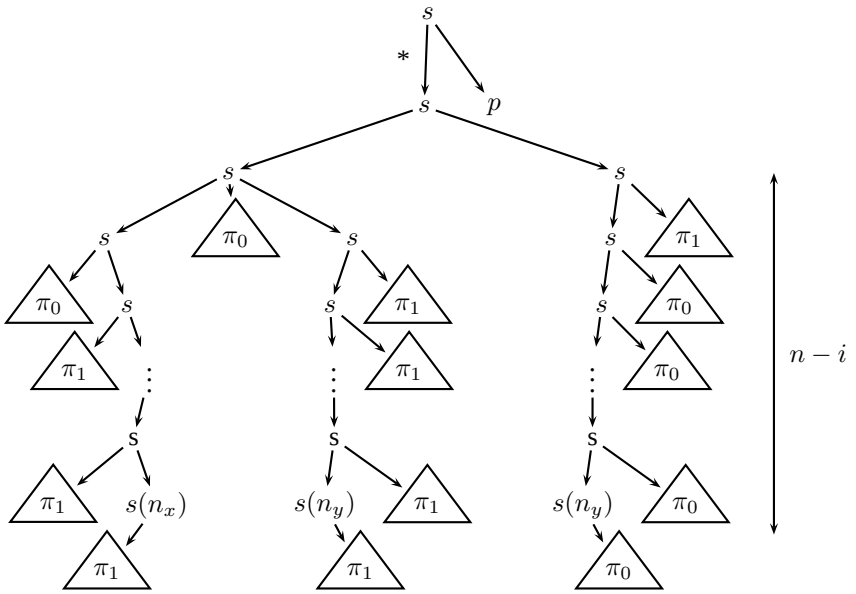


Figure 6: The pattern $\pi^i_{cons1}(x, y, z)$.

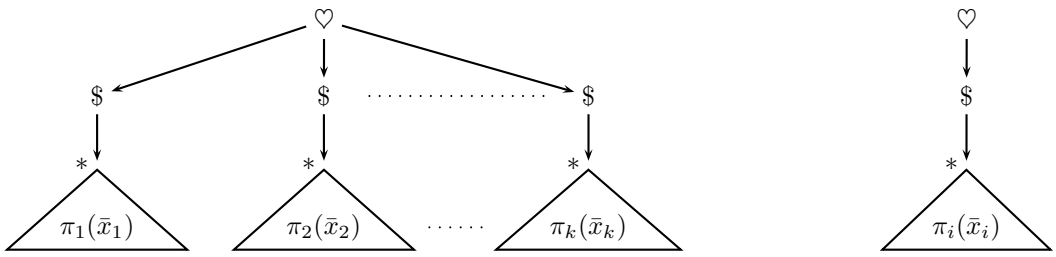


Figure 7: Patterns used in the proof of Lemma E.5.

The first part of the formula ensure that the nodes $x_a, y_b, z_c, x_d, y_e, z_f$ represent the same triple of consecutive skeleton leaves (in the shape corresponding to π^i_{consj}) in two consecutive tree t_c. The second part of the formula ensure that the a, b, c, d, e, f are attached to the leaves.

Notice that to cover all cases, we really need to consider the union of such queries for all i and j.

The formula $\varphi_{\mathcal{M}}$ is defined as the union of $\varphi_{abc,def}$ for all forbidden pairs of triples of letters (abc, def). The size of $\varphi_{\mathcal{M}}$ is polynomial in the size of the machine $\mathcal{M}$ and the length n of the word w. By construction, a data tree satisfies the constraints given by $\mathcal{R}$ and the negation of $\varphi_{nokey} \cup \varphi_{\mathcal{M}}$ iff it is an encoding of an accepting run of the machine on the word w. □

Notice that, if the class of queries in Proposition E.4 would have allowed wildcard, we could have done a brief reduction from the problem in Theorem 6 from [10] using φ_{nokey}. Indeed using wildcard, we can express this validity problem in terms of query containment. More precisely, in this case, we can show that given a conjunctive query q from CQ(Child,Child+) [10] and a regular language L, one can build in polynomial time a conjunctive tree query q_t so that q is valid w.r.t. L iff $q_\emptyset \subseteq_L q_t$.

The following lemma explains a trick to reduce UCQ containment to CQ containment. Similar lemmas can be found in [31] and [10]. We adapt it to our purposes.

LEMMA E.5. *Let $q_0, q_1, \cdots q_k \in CQ(\Downarrow, \Rightarrow, \sim)$ be conjunctive queries without free variables and $\mathcal{A}$ a NTA defining a regular tree language over the finite alphabet Σ. One can compute in polynomial time two queries q' and $q^\cup$ from CQ$(\Downarrow, \Rightarrow, \sim)$ and a NTA $\mathcal{A}^\cup$ over the alphabet $\Sigma \uplus \{\$, \#, \heartsuit, \spadesuit\}$ such that:*

$$q_0 \subseteq q_1 \cup \ldots \cup q_n \text{ w.r.t } \mathcal{A} \quad \text{iff} \quad q' \subseteq q^\cup \text{ w.r.t. } \mathcal{A}^\cup$$

Notice that the proof below only requires $\Downarrow$. The same lemma holds for less expressive classes of queries assuming they allows both $\downarrow$ and $\downarrow^*$ navigation. In particular the lemma holds for the classes CQ$(\Downarrow, \Rightarrow)$ and CQ$(\Downarrow, \sim)$.

PROOF. Let $q_0, q_1, \ldots, q_k \in$ CQ$(\Downarrow, \sim)$ be conjunctive queries without free variables and $\mathcal{A}$ a NTA defining a regular tree language over the finite alphabet Σ.

Without loss of generality, we can assume that for each i the query q_i is of the form $\exists \bar{x}_i \ \pi_i(\bar{x}_i)$. A query consisting in the conjunction of several patterns can be rewritten into the disjunction for every letter a in Σ of the query obtained by linking all patterns below a a-node using the relation $\downarrow^*$.

Moreover, modulo renaming, we can assume that the sets of variables used in the queries are all different.

We now explain how to build two queries q' and $q^\cup$ from CQ$(\Downarrow, \sim)$ and a NTA $\mathcal{A}^\cup$ over the alphabet $\Sigma \uplus \{\$, \#, \heartsuit, \spadesuit\}$ such that:

$$q_0 \subseteq q_1 \cup \ldots \cup q_k \text{ w.r.t } \mathcal{A} \quad \text{iff} \quad q' \subseteq q^\cup \text{ w.r.t. } \mathcal{A}^\cup$$

Figure 7 describes some patterns that we will need in the reduction. Recall that the labels $\heartsuit$ and \$ are new and do not occur in the alphabet Σ. Intuitively the pattern π_G is the conjunction of the patterns π_1 to π_k under some special nodes $\heartsuit$ and \$. Similarly, each pattern π_{q_i} is obtained from the pattern π_i by adding the nodes $\heartsuit$ and \$ at the root.

The queries q' and $q^\cup$ are built from the patterns π_G and π_{q_i} as shown in Figure 8. Again, $\#$ and $\spadesuit$ are new labels. The variables used in the copies of π_G and π_{q_i} are the one used in figure 7.

The automaton $\mathcal{A}^\cup$ checks the following properties.

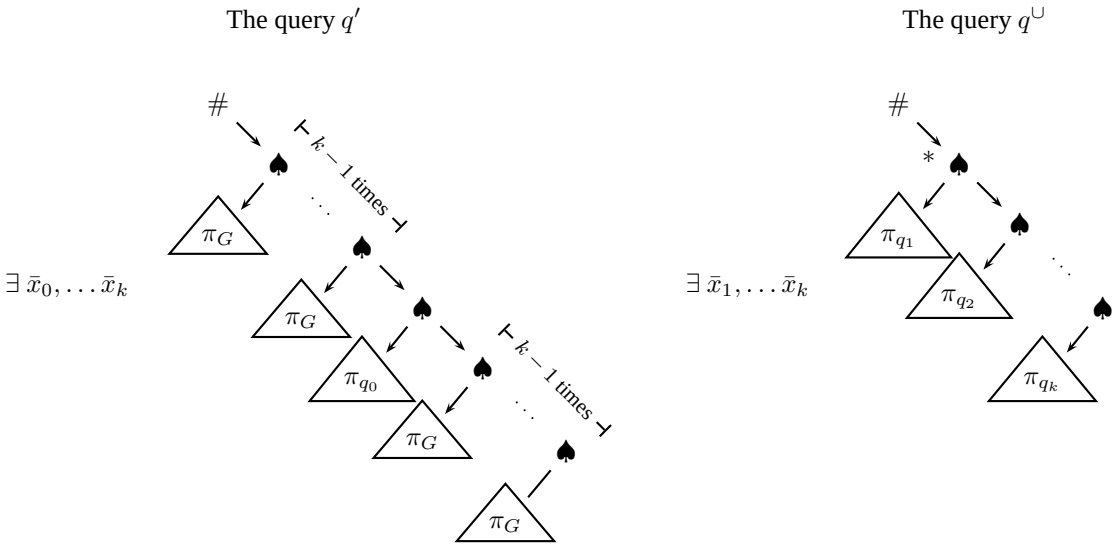


Figure 8: Queries q' and $q^{\cup}$ from the proof of Lemma E.5.

1. There are exactly $2k-1$ nodes with label ♠ and $2k-1$ nodes with label ♡.
2. There are exactly $k \cdot 2(k-1)+1$ nodes with label \$.
3. The root has label $\#$ and has exactly one child. This child has label ♠.
4. Each ♠-labeled node, except one, has one ♠-labeled child.
5. Each ♠-labeled node has exactly one child labeled ♡.
6. Each ♡-labeled node has only \$-children. Moreover the ♡-labeled node that is child of the kth ♠-labeled node, counted from the root has exactly one child, labeled \$. We call this the *distinguished* ♡-labeled node.
7. Each \$-labeled node has exactly one child.
8. The tree rooted at the grandchild of the distinguished ♡-labeled node is accepted by $\mathcal{A}$.

It remains to prove that

$$q_0 \subseteq q_1 \cup \ldots \cup q_k \text{ w.r.t } \mathcal{A} \quad \text{iff} \quad q' \subseteq q^{\cup} \text{ w.r.t. } \mathcal{A}^{\cup}$$

($\Rightarrow$) Assume that $q_0 \subseteq q_1 \vee \cdots \vee q_k$ w.r.t. $\mathcal{A}$. Consider a data tree t which satisfies the regular constraints given by $\mathcal{A}^{\cup}$ and the query q'. Let denote by v a valuation of the variables $\bar{x} = \{\bar{x}_0, \ldots, \bar{x}_k\}$ corresponding to a matching of the query q'. Let $s_1, \ldots, s_k, \ldots, s_{2k-1}$ be the ♠-labeled nodes of t, ordered by increasing distance from the root. For $j \in \{1, \ldots, 2k-1\}$, let t_j be the *tree* rooted in the ♡-labeled child of s_j. For each $j \in \{1, \ldots, 2k-1\} - \{k\}$, we note that since $t_j, 0, v \models \pi_G(\bar{x})$, we also have $t_j, 0, v \models \pi_{q_i}(\bar{x}_i)$ for every $i \in \{1, \ldots, k\}$. Let's now look at the subtree t_k and denote by t'_k the subtree rooted in the sole grandchild of the root of t_k. Since $t_k, 0, v \models \pi_{q_0}(\bar{x}_0)$ we have $t'_k, 0 \models q_0$. Moreover by definition of $\mathcal{A}^{\cup}$, the tree t'_k is recognized by $\mathcal{A}$. We conclude that $t'_k \models q_1 \vee \cdots \vee q_k$ i.e. there is $i \in \{1, \ldots, k\}$ and a valuation v' of the variables $\bar{x}_i$ such that $t'_k, v' \models \pi_i(\bar{x}_i)$. This, in turn, means that $t_k, 0, v' \models pi_{q_i}(\bar{x}_i)$.

We can show construct a matching of $q^{\cup}$ in t. The patterns $\pi_{q_1}, \ldots, \pi_{q_{i-1}}$ match respectively in $t_{k-i+1}, \ldots, t_{k-1}$ using the valuation v, π_{q_i} matches in t_k using the valuation v', and $\pi_{q_{i+1}}, \ldots, \pi_{q_k}$ match respectively in $t_{k+1}, \ldots t_{2k-i}$ using the valuation v. As the sets of variables $\bar{x}_j$ are disjunct and by construction of $q^{\cup}$, this matching is a witness of $t \models q^{\cup}$.

($\Leftarrow$) Assume, on the other hand, that $q_0 \not\subseteq q_1 \vee \cdots \vee q_k$ w.r.t. $\mathcal{A}$. Let p be a tree which satisfies the regular constraints given by $\mathcal{A}$, the query q_0 but does not satisfies $q_1 \vee \ldots \vee q_k$. Let t be a tree satisfying $\mathcal{A}^{\cup}$ and q', and define t_k and t'_k as above.

Replace t'_k by p in t. The resulting tree t_p still satisfies $\mathcal{A}$ and q', since p is accepted by $\mathcal{A}$ and q_0 is satisfied in p. But since no q_i, for $i \in \{1, \ldots, k\}$ is satisfied in p, there is no matching of $q^{\cup}$ in t_p. Thus $q' \not\subseteq q^{\cup}$ w.r.t. $\mathcal{A}^{\cup}$. $\square$

F. PROOFS FROM SECTION 8

Theorem 8.1. *Containment of BCCQs with data comparisons, i.e., the problem* $\mathrm{BCCQ}_{\subseteq}(\Downarrow, \Rightarrow, \_, \sim)$*, is undecidable.*

This theorem is a corollary of the following result:

THEOREM F.1. *Satisfiability of* $\mathrm{BCCQ}(\Downarrow, \Rightarrow, \_\ \sim)$ *is undecidable.*

PROOF. The proof follows that same lines as the proof of Theorem 15 of [10] and is by reduction from *Post's Correspondence Problem* (PCP), which is known to be undecidable. An *instance* of PCP over (finite) alphabet Σ is a sequence $(w_1, u_1), \ldots, (w_n, u_n)$ of pairs, where w_i, u_i are non-empty words over Σ^+, for all $i \in \{1, \ldots, n\}$. An instance has a *solution* if there exists an $m \in \mathbb{N}$ and $i_1, \ldots, i_m \in \{1, \ldots, n\}$ such that $w_{i_1} \ldots w_{i_m} = u_{i_1} \ldots u_{i_m}$.

Given an instance $R = (w_1, u_1), \ldots, (w_n, u_n)$ of PCP over alphabet Σ, we can construct a BCCQ Q that has a solution if and only if R has a solution.

The present reduction can be done using either only the relations $\downarrow$, $\downarrow^*$, and $\rightarrow$ or using only the relations $\downarrow$, $\rightarrow$, and $\rightarrow^*$. We present the first version here.

Let $R = (w_1, u_1), \ldots, (w_n, u_n)$ be an instance of PCP over the alphabet $\Sigma = \{a_1, \ldots, a_k\}$

Encoding of a solution first into a sequence of data and then into data trees.

Let $i_1, \ldots, i_m \in \{1, \ldots, n\}$ be a solution of R. We denote by w the word $w_{i_1} \ldots w_{i_m} = u_{i_1} \ldots u_{i_m}$.

We first explain how to encode a solution of a PCP instance into a special sequence of data values. This is built from several (all different) data values. We first list and name these data values:

- the data values $d_1 \cdots d_n$ where n is the number of different pairs of words in the instance R;
- the data values $d_{n+1} \cdots d_{n+k}$ representing the letters from the alphabet Σ;
- the data $C_1 \cdots C_m$ where m is given by number of pairs used to build the solution;
- the data $P_1 \cdots P_{|w|}$ are used to index positions in the word w;
- the data d_0 used as a separator in the encoding.

The corresponding encoding the solution is the sequence $d_0.E_R.d_0.E_w.d_0.E_u.d_0$ where E_R, E_w and E_u are defined as follows.

- $E_R := d_1 \cdots d_n.d_{n+1} \cdots d_{n+k}$ represents the instance R.
- The word $w_{i_1} \ldots w_{i_m}$ is represented by the following sequence of length $2.m + 2.|w|$:

  $$E_w := enc(w_{i_1}) \ldots enc(w_{i_m})$$

  where $enc(w_{i_j})$ is a sequence of length $2 + 2.|w_{i_j}|$ starting by $d_{i_j}.C_j$ and followed by the sequence obtained by replacing each position of w_{i_j} with the data representing its label followed by the data from $P_1 \ldots P_{|w|}$ representing the index of this position in the word w.
- Similarly, the word $u_{i_1} \ldots u_{i_m}$ is represented as the sequence:

  $$E_u := enc(u_{i_1}) \ldots enc(u_{i_m})$$

  where $enc(u_{i_j})$ is defined the same way as $enc(u_{i_j})$.

We extend this encoding to data trees. A data tree is an encoding of the solution iff it has a single branch, the corresponding sequence of data is of the form described above, the root (i.e. the first occurrence of data d_0) is labeled s, the second occurrence of d_0 is labeled \$ and the two last occurrences of d_0 are labeled $\#$ and all other nodes are labeled by letters from $\Gamma = \mathcal{L} \backslash \{s, \$, \#\}$ (in particular, the path is labeled by $\$\Gamma^*\$\Gamma^*\#\Gamma^*\#$).

We now construct a query $Q \in \mathrm{BCCQ}(\Downarrow, \rightarrow, \_\ \sim)$ without free variables such that a data tree satisfies the query Q iff it is an encoding of a solution of the PCP instance R.

The definition of the query Q.

In the remainder of the proof, we will construct several sub-formulas; and Q is a conjunction of them.

In the subqueries below, we sometimes simply write a instead of $a(x)$ to improve readability. We do this if we put no constraints on x (i.e., no equality and no inequality). We do this similarly for the symbols \$, $\#$, and the wildcard _. Similarly, for the variables x, y, z and their indexed versions, we write (x) as a shorthand for $\_(x)$ denoting a node with data value bound to x. We also often omit square brackets in queries to improve readability.

We first establish that PCP encodings are string-shaped. To this end we add

$$\neg\left(\_[\_ \to \_]\right)$$

to Q, which states that no node has more than one child. This is the only place in the proof where we use the relation $\to$. We now say that there is a special label s that occurs exactly once in any tree that matches Q, i.e., we add the following conjunction to Q:

$$s \wedge \neg\big(s/\_//s\big)$$

To ensure that s occurs at the root we add:

$$\neg(\_/s)$$

Using similar queries we can enforce the label \$ to occur exactly once and the label $\#$ to occur exactly twice.

We now add the query

$$\exists x_0 \cdots x_{n+k} \;\; s(x_0)/(x_1)/\cdots/(x_{n+k})/\$//\#(x_0)/\_\,//\#(x_0) \wedge \bigwedge_{i\neq j\in\{0,\ldots,n+k\}} x_i \neq x_j$$

which fixes the occurrences of \$, and $\#$ together and the fact that the $n+k$ nodes below s have pairwise different data values which are also different from the data of s. These data $x_0 \cdots x_{n+k}$ corresponds to the data $d_0 \cdots d_{n+k}$ in the definition of the encoding.

In the remainder of the proof, we say that the path is of the form $sL\$P_1\#P_2\#$ and use L, P_1 and P_2 to refer to the different part of the path. Also, if a position in a tree is below the node \$ and has a data value x_i with $i \in I := \{1, \ldots n\}$ then we will interpret it as a position that carries the label i. If it has a data value x_i with $i \in \{n+1, \ldots, n+k\}$ the we will interpret it as a position that carries the label $a_i \in \Sigma$. We refer to the data values $\{x_1, \ldots, x_{n+k}\}$ as *label values*. All other data values will be referred to as *non-label values*.

We now need to ensure the proper structure of the paths P_1 and P_2 which must encode non-empty data strings as explained in the the definition of the encoding. To this end, P_1 and P_2 will be of even length; carry a label value on every odd position and carry a non-label value on every even position. To this end, we say:

$$\bigvee_{i=1}^{n+k} \;\; \exists x_1 \cdots x_{n+k} \;\; s/(x_1)/\cdots/(x_{n+k})/\$/(x_i)$$

(the first position of P_1 has a label value)

$$\bigvee_{i=1}^{n+k} \;\; \exists x_1 \cdots x_{n+k} \;\; s/(x_1)/\cdots/(x_{n+k})/\$//\#/(x_i)//\#$$

(the first position of P_2 has a label value)

$$\exists x, x_1 \cdots x_{n+k} \;\; s/(x_1)/\cdots/(x_{n+k})/\$/\_/(x) \wedge \bigwedge_{i=1}^{n+k} x \neq x_i$$

(the second position of P_1 has a non-label value)

$$\exists x, x_1 \cdots x_{n+k} \;\; s/(x_1)/\cdots/(x_{n+k})/\$//\#/\_/(x)//\# \wedge \bigwedge_{i=1}^{n+k} x \neq x_i$$

(the second position of P_2 has a non-label value)

Similarly, we express that the last positions of P_1 and P_2 have non-label values. Furthermore, we add

$$\bigwedge_{i=1}^{n+k} \neg \quad \exists y, x_1 \cdots x_{n+k} \quad \Big(s/(x_1)/\cdots/(x_{n+k})/\$/\_//(x_i)/\_/(y)//\#/\_//\# \wedge \bigwedge_{j=1}^{n+k} y \neq x_j \Big)$$

(if some position ℓ of P_1 has a label value then position $\ell + 2$ has a label value)

$$\bigwedge_{i=1}^{n+k} \neg \quad \Big(\exists y, x_1 \cdots x_{n+k} \quad s/(x_1)/\cdots/(x_{n+k})/\$/\_//(y)/\_/(x_i)//\#/\_//\# \wedge \bigwedge_{j=1}^{n+k} y \neq x_j \Big)$$

(if some position ℓ of P_1 has a non-label value then position $\ell + 2$ has a non-label value)

Similarly, we add the two above conditions for P_2.

So, all solutions to Q are of the form $sL\$P_1\#P_2\#$ in which P_1 and P_2 encode data strings. We now refine our notation for the remainder of the proof. We define

- W: the string over alphabet $\Sigma \cup I$ obtained from P_1 by considering the concatenation $y_1 \dots y_m$ of all label values and replacing each y_i by $i \in I$ if $y_i \in \{x_1, \dots, x_n\}$ and y_i with $a_j \in \Sigma$ if $y_i = x_{n+j}$.
- U: obtained from P_2 analogously as W is obtained from P_1.
- For a string V over alphabet $\Sigma \cup I$, we denote by $V_{|\Sigma}$ the Σ-string obtained from V by deleting all letters from I (and analogously for $V_{|I}$).

So, in the new notation, we are looking for trees of the form $sL\$P_1\#P_2\#$ that satisfy, among other conditions, that $W_{|I} = U_{|I}$ and $W_{|\Sigma} = U_{|\Sigma}$.

Next in the proof, we want to express that

(LAB): the string $W\#U\#$ should match the regular expression

$$\big((i_1 \cdot w_{i_1}) + \cdots + (i_n \cdot w_{i_n})\big)^+ \cdot \# \cdot \big((i_1 \cdot u_{i_1}) + \cdots + (i_n \cdot u_{i_n})\big)^+ \cdot \#$$

where each $i_j \in I$ and the words w_{i_j} and u_{i_j} are the correct ones from the PCP instance.

The idea is that we do this by saying that the first position from P_1 (resp., P_2) must be a label value $i \in \{1, \dots, n\}$ and by excluding patterns that are not allowed to occur.

$$\bigvee_{i=1}^{n} \quad \exists x_1 \cdots x_n \quad s/(x_1)/\cdots/(x_n)//\$/(x_i)$$

(the first label value from P_1 is an index from $\{1, \dots, n\}$)

$$\bigvee_{i=1}^{n} \quad \exists x_1 \cdots x_n \quad s/(x_1)/\cdots/(x_n)//\$//\#/(x_i)//\#$$

(the first label value from P_2 is an index from $\{1, \dots, n\}$)

We now say that every index i from $\{1, \dots, n\}$ in W must be followed by w_i. More precisely, we forbid all iw for

- w shorter than w_i but not a prefix of w_i;
- $w = w'\#$ where w' is a real prefix of w_i; and
- $w = w_i a$ with $a \notin \{1, \dots, n, \#\}$.

To this end, we define encodings of words $w = \sigma_1 \cdots \sigma_m$ over alphabet $\{1, \dots, n\} \cup \{a_1, \dots, a_k\}$ as pattern, that is,

$$\pi_w := (x_{j_1})/\_/(x_{j_2})/\_/\cdots/\_/(x_{j_m})$$

where, for each $\ell = 1, \dots m$, we have that

- if $\sigma_\ell = \#$ then $j_\ell = 0$,
- if $\sigma_\ell = i \in \{1, \dots, n\}$ then $j_\ell = i$, and

- if $\sigma_\ell = a_p$ then $j_\ell = n + p$

Then we express the above conditions for P_1 as

$$\bigwedge_{i=1}^{n} \bigwedge_{w} \neg \quad \exists x_1 \cdots x_{n+k} \quad s/(x_1)/\cdots/(x_{n+k})/\$//\_/\_(x_i)/\_/\pi_w/\_//\#\_//\#$$

where w and π_w are as mentioned above. We express the conditions (LAB) for P_2 similarly.

We show how to put constraints on the "data"-parts of P_1 and P_2 to ensure the following.

(ENC1): no non-label data value in $P_1\#P_2\#$ appears more than twice,
(ENC2): the sequence of non-label values associated to $W_{|I}$ matches the sequence of non-label values associated to $U_{|I}$,
(ENC3): the sequence of non-label values associated to $W_{|\Sigma}$ matches the sequence of non-label values associated to $U_{|\Sigma}$,
(ENC4): two positions from W and U associated to the same non-label values must have the same label value.

This will enforce that $W_{|\Sigma} = U_{|\Sigma}$ and $W_{|I} = U_{|I}$.

We enforce (ENC1) by:

$$\neg \quad \exists x, x_1 \cdots x_{n+k} \quad \big(s/\_(x_1)/\cdots/\_(x_{n+k})/\$/\_//\_(x)//\_/\_(x)//\_/\_(x) \wedge \bigwedge_{i=1}^{n+k} (x \neq x_i)\big)$$

(There do not exist three different nodes with data value x that does not occur in the $x_1, \ldots, x_{n+k}$.)

(ENC2) We add to Q a query that enforce that the first symbol from $W_{|I}$ has the same associated non-label value as the first symbol from $U_{|I}$. Notice that these positions necessarily carry label values from I.

$$\exists x \;\; s//\$/\_/\_(x)//\#/\_/\_(x)//\#$$

For an $\ell \in \mathbb{N}$, let dist_ℓ be the sub-pattern that matches a path of nodes of length ℓ, i.e.,

$$\text{dist}_\ell = \_/\cdots/\_$$

with ℓ occurrences of _.

For each $i \in \{1, \ldots, n\}$, let $m_i = |w_i|$ and $m'_i = |u_i|$, and define

$$\bigwedge_{i=1}^{n} \neg \quad \exists x, x_1 \cdots x_{n+k} \quad \big(s/\_(x_1)/\cdots/\_(x_{n+k})/\$//x_i/(x)/\text{dist}_{2m_i+1}/(y_1)//\#//(x_i)/(x)/\text{dist}_{2m'_i+1}/(y_2) \wedge y_1 \neq y_2\big)$$

The above query makes sure that if two I-positions have the same data value the same holds for their I-successors. Altogether this ensures the condition (ENC2).

(ENC4) We now want to express that, if two positions in $U \cdot W$ have the same data value, then they have the same label:

$$\bigwedge_{i \neq j \in \{1,\ldots,n+k\}} \neg \quad \exists x, x_1 \cdots x_{n+k} \quad s/(x_1)/\cdots/(x_{n+k})/\$//(x_i)/(x)//(x_j)/(x)$$

(It is not possible that two positions have the same non-label value and their parents have different label values.)

These last three queries ensure that $W_{|I} = U_{|I}$.

(ENC3) It remains to define the subqueries that ensure (ENC3). First, we make sure that the first letters in $W_{|\Sigma}$ and $U_{|\Sigma}$ have the same data value:

$$\exists y, x_1 \cdots x_{n+k} \quad s/(x_1)/\cdots/(x_{n+k})/\$/\_/\_/\_/(y)//\#/\_/\_/\_/(y)//\#$$

Next, we ensure that if positions x_1 and y_1, taken from the first and the second half of the string, respectively, have labels from Σ and the same data value, and their successors also have labels from Σ, then the successors have the same data value. Thus, for each ordered tuple $(a, b, c) \in \Sigma^3$, we define

$$\neg \quad \exists x, y, z, x_1 \cdots x_{n+k} \quad \big(s/(x_1)/\cdots/(x_{n+k})/\$//(x_a)/(x)/(x_b)/(y)//\#//(x)/(x_c)/(z) \wedge y \neq z\big)$$

We must, however, also allow for the cases that some of the Σ-nodes are followed by a node with a label from I. Thus, for every (a, b, c) from Σ and every i, j from I, we get

$$\neg \quad \exists x, y, z, x_1 \cdots x_{n+k} \quad \big(s/(x_1)/\cdots/(x_{n+k})/\$//(x_a)/(x)/(x_i)/\_/(x_b)/(y)//\#//(x)/(x_c)/(z) \wedge y \neq z\big)$$

(Case of a-labeled node followed by an i)

$$\neg \quad \exists x, y, z, x_1 \cdots x_{n+k} \quad \big(s/(x_1)/\cdots/(x_{n+k})/\$//(x_a)/(x)/(x_b)/(y)//\#//(x)/(x_i)/\_/(x_c)/(z) \wedge y \neq z\big)$$

(Case of c-labeled node followed by an i)

$$\neg \quad \exists x, y, z, x_1 \cdots x_{n+k} \quad \big(s/(x_1)/\cdots/(x_{n+k})/\$//(x_a)/(x)/(x_i)/\_/(x_b)/(y)//\#//(x)/(x_j)/\_/(x_c)/(z) \wedge y \neq z\big)$$

(Case of both nodes followed by an index.)

This concludes the reduction. □

Theorem 8.2. *The containment problem for CQ$(\Downarrow, \sim)$ queries under schema is undecidable.*

We first prove that under schema constraints $\mathrm{UCQ}_{\subseteq}(\Downarrow, \sim)$ is undecidable. Using Lemma E.5, we can get the lower bound for $\mathrm{CQ}_{\subseteq}(\Downarrow, \sim)$.

PROPOSITION F.2. *The containment problem for* $\mathrm{UCQ}(\Downarrow, \sim)$ *queries under schema is undecidable.*

We prove this result by adapting the proof of the undecidability for satisfiability of data tree patterns formulas from Theorem 7 in [16]. Also Notice that, for clarity reason, the patterns presented in the proof are using wildcard. We explain how to get rid of them at the end of the proof.

PROOF. We prove the undecidability by reducing the halting problem of two-counter machines (or Minsky machines) to our problem. Given a machine $\mathcal{M}$, we define a regular language $\mathcal{R}$ and a finite union of conjunctive queries φ such that any data tree satisfying the constraints given by $\mathcal{R}$ and the negation of φ is the encoding of an accepting run of the machine.

The machine $\mathcal{M}$ has no accepting run iff $q_\emptyset \subseteq_{\mathcal{R}} \varphi$ where $q_\emptyset$ is the trivial query which is always true.

In the following, we define precisely the encoding of accepting runs into a data trees and explain how to build the language $\mathcal{R}$ and the query φ.

A two-counter machine $\mathcal{M}$ is a finite state machine equipped with two counters (r_a and r_b) initially set to zero. A transition of the machine non-deterministically increments one of the counters and changes its current state; decrements one of the counters and changes its current state (with the restriction that the value of a counter cannot be negative); or checks whether a counter is zero and changes state depending on the result.
A run of the machine is a sequence $c_0\delta_1 c_1 \cdots \delta_n c_n$ where each c_i are configurations of the machine, δ_i are transitions and the evolution of the current state and the counter values is consistent with the transitions along the run. A run is accepting if it starts from the initial state with both counters being zero and end up in an accepting state. Deciding whether a two-counter machine has an accepting run is undecidable.

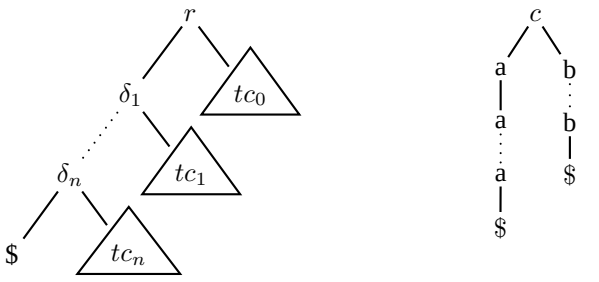
Let us define the finite alphabet $\Sigma = \{r, c, a, b, \$\} \cup \{\delta | \delta \text{ represents a transition of the machine}\}$. Without loss of generality we can assume that the information in δ for zero-test transitions includes the result of the test so that given such δ, we know what is the next state of the machine and whether the checked-counter should be zero or strictly positive.

A data tree is an encoding of a run $c_0\delta_1 c_1 \cdots \delta_n c_n$ is respect the following constraints:

The global structure of the tree is described in Figure 9(a). Under the root labeled r, the tree contains a branch where nodes are labeled by the transitions $\delta_1, \ldots, \delta_n$ ending with a leaf labeled \$. Each of these nodes (but the leaf) branches to a subtree encoding the value of the counters in the corresponding configuration. The root of this subtree is labeled c and it is made of two branches (see Figure 9(b)). If the counter first counter has value r_a, one branch is a sequence of $r_a + 1$ nodes labeled a and ends with a final \$-leaf. Similarly the other branch is a sequence of $r_b + 1$ nodes labeled b and ends with a \$-leaf where r_b is the value of the second counter.

The data values will allow us to control the evolution of the counters between consecutive configurations. In order to do so, we need to guarantee a certain structure and continuity of the values in the tree. (1) We impose that within any a-branch (resp b-branch), each a-node (resp. b-node) has a different data value. (2) Given two consecutive a-branches (resp. $b - branches$) containing respectively n and m nodes with $n \leq m$, the sequence of data attached to the first n nodes is the same in both a-branches (resp. b-branches). [2]

[2]The only relevant values will be the one in a-nodes and b-nodes. In particular, we don't impose any data-constraints for the other nodes. Thus we have several possible encodings of one run but we know that each accepting run has at least one encoding



(a) The structure of the tree (b) The structure of tc_i.

Figure 9: The structure of an encoding of a run

We now need to build a regular tree language $\mathcal{R}$ using the finite alphabet Σ and a union of queries φ such that a data tree satisfies the constraints given by $\mathcal{R}$ and the negation of φ iff it is an encoding of an accepting run of the machine.

The language $\mathcal{R}$ ensures the general structure of the tree, the labeling of the nodes. It also ensures that the sequence of transitions respects the machine's rules in terms of succession of control states, initial and final configurations. Additionally, we encode in $\mathcal{R}$ the behaviour of zero-test transitions (a counter is zero iff the corresponding branch has exactly one a-node (resp. b-node)). All these constraints are regular properties of the trees so we can encode them into a regular tree automaton.

The union of queries $\varphi = \varphi_a \cup \varphi_b$ is used to ensure the constraints on data values and the correct evolution of a-branch (resp b-branch) between consecutive configurations (that is the evolution of the counters along the corresponding run).

We show how to build φ_a corresponding to the first counter. The formula φ_b can be constructed similarly. Intuitively φ_a describes all the behaviour that violates the proper evolution of the a-branches.

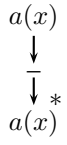
The evolution of the a-branches is correct in a data tree that satisfies the regular constraints $\mathcal{R}$ iff:

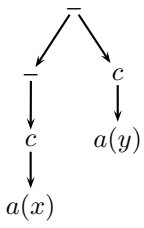
1. in any configuration tree, the data values in the a-branches are all different and if two a-nodes correspond to the same position in two successive configuration trees, they share the same data value;

2. and in between two consecutive configuration trees, the length of the a-branches change according to the current transition.

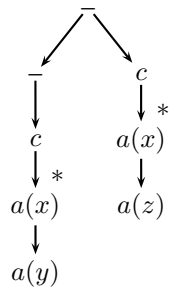
The first item can be enforced by negating the formula $\varphi^a_{struct} := q_a \cup q_{first} \cup q_{cont}$ where each query is defined below. Forbidding q_a ensures that in any configuration tree, the data values in the a-branches are all different and forbidding the query $q_{first} \cup q_{cont}$ ensure the continuity of the sequences of data in between two consecutive configuration trees.

For each query, we give also a graphical representation that may be easier to parse.

$q_a := \exists x \;\; a(x)/\_//a(x)$ can also be represented as $\exists x$ 

$q_{first} := \exists x, y \;\; \_/[\_/c/a(x), c/a(y)] \;\wedge\; x \neq y$ can also be seen as $\exists x, y$  with $x \neq y$

$q_{cont} := \exists x, y, z \ \_/[\_/c//a(x)/a(y), c//a(x)/a(z)] \ \wedge \ y \neq z$ can also be seen as $\exists x, y, z$

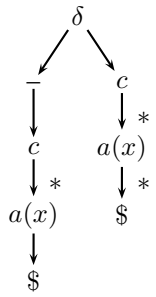


with $y \neq z$

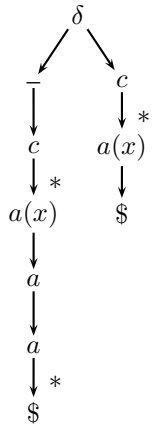
The second item can be enforced by forbidding the union of φ^a_δ for all transition δ. Recall that the behaviour of zero-test transitions has already been encoded into the regular language $\mathcal{R}$. There are three cases : either δ increments the first counter, either it decrements the first counter or it lets it unchanged.

First assume that δ is a transition that increments the first counter. Forbidding the formula $\varphi^a_\delta := q^{a\leq 0}_\delta \cup q^{a\geq 2}_\delta$ defined below ensures that in between two consecutive configurations driven by the transition δ the first counter has been incremented by one. Intuitively $q^{a\leq 0}_\delta$ is true if the first counter has been decremented or is unchanged and $q^{a\geq 2}_\delta$ is true if the first counter has been incremented at least twice.

$\pi^{a\leq 0}_\delta(x) := \delta/[\_/c//a(x)/\$, c//a(x)//\$]$ can also be seen as



$\pi^{a\geq 2}_\delta(x) := \delta/[\_/c//a(x)/a/a//\$, c//a(x)/\$]$ can also be seen as



Similarly if δ is a transition that decrements the first counter. Forbidding the formula $\varphi^a_\delta := q^{a\geq 0}_\delta \cup q^{a\leq 2}_\delta$ defined below ensures that in between two consecutive configurations driven by the transition δ the first counter has been decremented by one. Intuitively $q^{a\geq 0}_\delta$ is true if the first counter has been incremented or is unchanged and $q^{a\leq 0}_\delta$ is true if the first counter has been decremented at least twice.

$$\pi^{a\geq 0}_\delta(x) := \delta/[\_/c//a(x)//\$, c//a(x)/\$] \quad \text{and} \quad \pi^{a\leq 2}_\delta(x) := \delta/[\_/c//a(x)/\$, c//a(x)/a/a//\$]$$

Finally if δ is a transition that neither increments or decrements the first counter, forbidding the formula $\varphi^a_\delta := q^{a\geq 1}_\delta \cup q^{a\leq 1}_\delta$ defined below ensures that in between two consecutive configurations driven by the transition δ the first counter hasn't changed. Intuitively $q^{a\geq 1}_\delta$ is true if the first counter has been incremented and $q^{a\leq 1}_\delta$ is true if the first counter has been decremented.

$$\pi^{a\geq 1}_\delta(x) := \delta/[\_/c//a(x)/a/\$, c//a(x)/\$] \quad \text{and} \quad \pi^{a\leq 1}_\delta(x) := \delta/[\_/c//a(x)/\$, c//a(x)/a/\$]$$

The formula φ_a is now defined as $\varphi^a_{struct} \cup_\delta \varphi^a_\delta$.

The last thing to explain is how to transform this formula into a formula where patterns do not use the wildcard. As the alphabet Σ of relevant labels in finite, this can be done by taking the union for each query that contains wildcard of all possible queries obtained by replacing the wildcards with some labels from Σ.

The formula φ_b can be defined the same way as a finite union of queries.

By construction, a data tree satisfies the regular constraints given by $\mathcal{R}$ and doesn't satisfies $\varphi_a \cup \varphi_b$ iff it is an encoding of an accepting run of the two-counter machine. □

Proposition 8.3. *The problem $CQ_{\subseteq}(\downarrow, \sim)$ is Π_2^p-hard.*

PROOF. We proceed by reduction from $\forall\exists 3CNF$ and simply adapt the proof of Theorem 4.2.

Given as input a formula $\varphi := \forall p_1 \ldots \forall p_l \exists q_1 \ldots \exists q_m \bigwedge_{1 \leq i \leq n} (l_{i1} \vee l_{i2} \vee l_{i3})$, we construct queries $q_{\neq}, q'_{\neq} \in \text{CQ}(\downarrow, \sim)$ such that:

$$\varphi \text{ is true} \quad \text{if and only if} \quad q_{\neq} \subseteq q'_{\neq}.$$

We use the same alphabet of labels and patterns Π_{ij}'s as in the proof of Theorem 4.2. The queries $q_{\neq}, q'_{\neq}$ are defined as follows:

- $q_{\neq} := \exists x_1 \ldots \exists x_l (\bigwedge_{1 \leq i \leq n, 1 \leq j \leq 3} \Pi_{ij} \wedge Val(0) \wedge Val(1) \bigwedge_{1 \leq k \leq l} Val(x_k))$
  and
- $q'_{\neq} := \exists y_1 \ldots \exists y_m \; (\bigwedge_{1 \leq i \leq m} Val(y_i) \wedge$
  $\bigwedge_{1 \leq j \leq n} \exists z \exists z' \; (Cl(c_j)/L(z)/P(0)/Oc(0)$
  $\wedge Cl(c_j)/L(z)/P(z')/Oc(1) \wedge z' \neq 0$
  $\wedge \bigwedge_{1 \leq k \leq m} Cl(d_j)/L(z_1)/Q(q_k)/Oc(y_k)))$

The reduction is a straightforward adaptation of the proof of Theorem 4.2. As before, any model T of the first query will be considered to encode a valuation of the p_i's. This time, the valuation will be defined as follows: if x_i is interpreted by 0 in T, then $v(p_i) = 0$, otherwise $v(p_i) = 1$. In this way, every possible valuation of the p_i's will be encoded by a different model of the first query. □