Monadic datalog and regular tree pattern queries

FILIP MAZOWIECKI, FILIP MURLAK, and ADAM WITKOWSKI, University of Warsaw

Containment of monadic datalog programs over trees is decidable. The situation is more complex when tree nodes carry labels from an infinite alphabet that can be tested for equality. Then, it matters whether the descendant relation is allowed or not: the descendant relation can be eliminated easily from monadic programs only when label equalities are not used. With descendant, even containment of linear monadic programs in unions of conjunctive queries is undecidable and positive results are known only for bounded-depth trees.

We show that without descendant containment of connected monadic programs is decidable over ranked trees, but over unranked trees it is so only for linear programs. With descendant it becomes decidable over unranked trees under restriction to downward programs: each rule only moves down from the node in the head. This restriction is motivated by regular tree pattern queries, a recent formalism in the area of ActiveXML, which we show to be equivalent to linear downward programs.

Categories and Subject Descriptors: H.2.4 [**Systems**]: Query processing

General Terms: Theory

Additional Key Words and Phrases: recursive queries, trees, semistructured data, containment problem

ACM Reference Format:
Filip Mazowiecki, Filip Murlak and Adam Witkowski. 2016. Monadic datalog and regular tree pattern queries. *ACM Trans. Datab. Syst.* V, N, Article XXXX (April 2016), 44 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Query languages based on first order logic, even extended with counting capabilities, like SQL, have limited expressive power due to their locality. To express non-local properties, recursion is used. A classical recursive query language is datalog, obtained by extending unions of conjunctive queries (UCQs) with a fixpoint operator; for instance, the existence of a path between nodes X and Y in a graph is well known not to be expressible in first order logic (nor in full SQL, see [Libkin 2003]), but is easily expressible with a datalog program:

$$\begin{aligned} &path(X,Y) \leftarrow E(X,Y)\,; \\ &path(X,Y) \leftarrow E(X,Z), path(Z,Y)\,. \end{aligned}$$

Unsurprisingly, the increased expressive power makes the analysis of queries very difficult. Consider the following classical example [Naughton 1989]:

$$\begin{aligned} \mathcal{P}\colon\ & buys(X,Y) \leftarrow likes(X,Y)\,; \\ & buys(X,Y) \leftarrow trendy(X), buys(Z,Y)\,. \end{aligned} \qquad \begin{aligned} \mathcal{P}'\colon\ & buys(X,Y) \leftarrow likes(X,Y)\,; \\ & buys(X,Y) \leftarrow trendy(X), likes(Z,Y)\,. \end{aligned}$$

The two programs are equivalent, but the second does not use recursion at all. Thus, by replacing a recursive call to the intensional predicate $buys$, with a call to the extensional predicate $likes$, readily available in the database, one can avoid costly recursive evaluation. One could also optimize queries by removing redundant rules or applying

other, more complicated heuristics. Any such heuristic will eventually need to check if the modified query is indeed equivalent to the original. This is not an easy task; after the following, seemingly innocent, modification the two programs are not equivalent any more (in fact, $\mathcal{Q}$ is not equivalent to any non-recursive program):

$$\mathcal{Q}\colon \begin{array}{l} buys(X,Y) \leftarrow likes(X,Y)\,; \\ buys(X,Y) \leftarrow knows(X,Z), buys(Z,Y)\,. \end{array} \qquad \mathcal{Q}'\colon \begin{array}{l} buys(X,Y) \leftarrow likes(X,Y)\,; \\ buys(X,Y) \leftarrow knows(X,Z), likes(Z,Y)\,. \end{array}$$

How to distinguish the two situations?

Background. Traditionally, static analysis of queries focuses on a more fundamental decision problem of *containment*, in which one asks if the results of one query are always contained in the results of another query; equivalence easily reduces to containment, and *vice versa* for query languages closed under union. For full datalog both these problems are in general undecidable [Shmueli 1993]. It is also undecidable if a given datalog program is equivalent to some non-recursive datalog program [Gaifman et al. 1993] (equivalence to a given non-recursive program is decidable [Chaudhuri and Vardi 1992]). The negative results for full datalog fuel interest in decidable fragments. Most importantly, containment is decidable for *monadic* programs, using only unary intensional predicates, with better complexity bounds for programs that are also *linear*, that is, have at most one use of an intensional predicate in each rule [Cosmadakis et al. 1988; Benedikt et al. 2012]; but interesting extensions have been proposed as well [Bonatti 2004; Calvanese et al. 2005].

A different way to decidability leads via a restriction of the class of possible database instances, typically to trees or tree-like structures. This can be justified either by the nature of the data—for instance, a representation of the hierarchical structure of a company—or by the data format itself, as is the case for XML or JSON. When the class of structures is restricted to finitely-labelled trees (or words), even satisfiability is undecidable (see e.g. [Abiteboul et al. 2013]), but for monadic programs we regain decidability of containment [Gottlob and Koch 2004; Frochaux et al. 2014]. This decidability result, however, does not carry over to a more adequate model of real-life tree-like data, in which tree nodes carry labels from an infinite alphabet that can be tested for equality. A recent paper [Abiteboul et al. 2013] shows that in this model even containment of linear monadic programs in unions of conjunctive queries is undecidable. Restriction to bounded-depth trees restores decidability, even for containment of arbitrary programs in monadic programs, but this is not very surprising: without sibling order datalog programs cannot make much of unbounded branching.

Focus. In this paper we aim beyond bounded-depth trees. We work with unordered trees over infinite alphabet; that is, the extensional predicates include label tests, label equality, child, and descendant, but not sibling order. This is in tune with a large proportion of the literature of the subject, but in real-life applications sibling order is meaningful and may be used in queries. Unfortunately, its presence quickly leads to undecidability; we discuss this in more detail in the concluding section of the paper. Moreover, we consider only *connected* programs; that is, all nodes mentioned in the program are connected by child or descendant predicates. This is not an uncommon assumption either [Cosmadakis et al. 1988; Gaifman et al. 1993]. A practical reason is that real-life programs tend to be connected (cf. [Bancilhon and Ramakrishnan 1986]). Programs that are not connected combine pieces of unrelated data, which corresponds to the *cross product*, an unnatural operation in the database context. It seems even more natural to assume connectedness in the tree-structured data model.

The starting point is the observation that for monadic programs it matters if the descendant relation is allowed or not. Indeed, the simplest idea of replacing each use

of descendant with its recursive definition in terms of child (analogous to the first example), does not give a monadic program. In fact, descendant can only be eliminated from monadic programs that use label equality locally: for instance, rule

$$P(X) \leftarrow Q(X), \mathit{child}(X,Y), X \sim Y, \mathit{desc}(Y,Z), R(Z)\,,$$

which compares labels of nodes connected with the child predicate, can be replaced with

$$\begin{aligned} P(X) &\leftarrow Q(X), \mathit{child}(X,Y), X \sim Y, R'(Y)\,,\\ R'(Y) &\leftarrow \mathit{child}(Y,Z), R'(Z)\,,\\ R'(Y) &\leftarrow \mathit{child}(Y,Z), R(Z)\,, \end{aligned}$$

but the same approach does not work for rule

$$P(X) \leftarrow Q(X), \mathit{child}(X,Y), X \sim Y, \mathit{desc}(Y,Z), R(Z), Y \sim Z\,,$$

which compares labels of nodes connected only with the descendant predicate. In fact, one cannot even express that a node has a descendant with the same label. As the undecidability result of [Abiteboul et al. 2013] uses the descendant predicate a lot, it does not carry over directly to *child-only programs*, that is, programs that do not use descendant. This is one of the two fragments we focus on.

For the case with descendant, we work with the class of *downward programs*, where each rule only moves down from the node in the head. While forbidding descendant is a natural restriction (pointed out as an open problem in [Abiteboul et al. 2013]), downward programs may seem exotic. But in fact, they were our initial point of interest. Our original motivation comes from the area of ActiveXML, where testing equivalence of ActiveXML systems amounts to solving the containment problem for regular tree pattern queries (RTPQs): essentially, regular expressions over the set of tree patterns using child and descendant axes and label equalities. In [Abiteboul et al. 2011] it is shown that the containment problem is decidable (in EXPTIME) for RTPQs with restricted use of label equalities, mimicking data comparisons allowed in XPath (an XML query language used widely in practice and extensively studied [Benedikt et al. 2008; Figueira 2009; Miklau and Suciu 2004; Neven and Schwentick 2006]). This result relies on Figueira's EXPTIME-completeness of satisfiability for RegXPath$(\downarrow, =)$, i.e., XPath with child axis and data equality, extended with Kleene star (descendant is expressible with child and Kleene star, unlike for monadic datalog) [Figueira 2009]. We show that RTPQs are equivalent to linear downward programs (a special case of a more general correspondence between monadic programs and a natural extension of RTPQs). Hence, decidability of the containment problem for linear downward programs immediately gives decidability of the unrestricted ActiveXML problem.

The use of label equalities, which was the source of difficulties in [Abiteboul et al. 2011], is precisely what distinguishes RegXPath from monadic datalog. Without label equalities, positive RegXPath (no negation) coincides with monadic datalog, and the correspondence extends to natural fragments of the two formalisms. Once label equalities are allowed, the expressive power of RegXPath falls short even of conjunctive queries: for instance, one cannot express in RegXPath (positive or not) that a given node has three descendants with the same label, whose parents have labels a, b, c. On the other hand, as we have seen, restricting to monadic programs also limits the way in which labels can be compared in datalog: an example of a property expressible in positive RegXPath$(\downarrow, =)$, but not in monadic datalog, is that the same label can be reached from a given node by going down through a-labelled nodes, and by going down through

b-labelled nodes. This can be expressed easily using binary intensional predicates:

$$P(X) \leftarrow Y \sim Z, desc_a(X,Y), desc_b(X,Z),$$
$$desc_\sigma(X,Y) \leftarrow child(X,X'), \sigma(X'), desc_\sigma(X',Y),$$
$$desc_\sigma(X,Y) \leftarrow child(X,Y)$$

with $\sigma = a, b$, but there is no way to compare the labels of the two descendants of X without passing them as arguments of the intensional predicates. Thus, in the presence of label equalities, the two formalisms diverge and neither upper bounds nor lower bounds carry over between them.

Results. Our results are summarized in Table I, where D-Datalog$(\downarrow, \downarrow^+)$ stands for downward programs, and Datalog$(\downarrow)$ stands for child-only programs. We are mostly interested in *unranked* trees, where nodes can have arbitrarily many children, but to provide a broader background for our main results, we also consider *ranked* trees, where the number of children is fixed, and words, which can be seen as unary trees.

	Unranked trees		Ranked trees	Words
	linear	non-linear	linear/non-lin.	linear/non-lin.
D-Datalog$(\downarrow, \downarrow^+)$	EXPSPACE	2-EXPTIME	UNDEC.	UNDEC.
Datalog$(\downarrow)$	2-EXPTIME-hard in 3-EXPTIME	UNDEC.	2-EXPTIME	PSPACE

Table I: Complexity of containment for monadic datalog fragments.

Our main contributions are the decision procedures for the containment of downward programs and linear child-only programs over unranked trees. In the latter case there is a gap between the upper and lower bound, but the problem becomes 2-EXPTIME-complete when restricted to containment in non-recursive programs. Rather unexpectedly, for non-linear child-only programs we lose decidability: it turns out that using non-linearity we can simulate the descendant predicate to the extent sufficient to repeat the reduction used for unrestricted programs. It is also unusual that for ranked trees and words the situation is essentially the opposite: for non-linear child-only programs containment is decidable, and for downward (even linear) programs it is not. (Due to slight discrepancies between data models, this result does not follow directly from [Abiteboul et al. 2013], and has to be reproved).

Techniques. The differences between the ranked and unranked case are reflected in the techniques used to show decidability. In the ranked case the proof strategy is rather standard:

(1) show that any tree that is a counter-example to containment can be relabeled so that it only uses labels from a fixed alphabet of exponential size;
(2) turn both programs to automata working over finitely-labelled trees and test them for containment of the recognized languages.

For words the complexity is lower because the bound on the alphabet in (1) is polynomial and the containment test is cheaper.

For unranked trees the approach is somewhat dual: instead of limiting the size of the alphabet, we make the labels in the counter-example as different as possible. More precisely, the proof strategy is as follows:

(1) show that any counter-example to containment can be turned into a canonical one (essentially, a universal model for one of the programs), which avoids merging witnessing nodes and reusing labels whenever possible;
(2) turn both programs to automata working over finitely-labelled trees encoding the canonical counter-examples and test them for containment of the recognized languages.

Interestingly, it is what makes one approach fail that makes the other work: bounding the alphabet size requires bounded branching, and not merging witnessing nodes requires unbounded branching.

The proofs of the upper bounds are self-contained, modulo standard properties of word and tree automata, plus a lesser-known result on alternating two-way automata [Cosmadakis et al. 1988; Vardi 1998]. Lower bounds use standard hard problems, or their easy variations.

Improvements. The exponential gap for linear child-only programs over unranked trees, which was already pointed out in the conference version of this paper, has been recently closed using rather different methods: the problem turns out to be 2-EXPTIME-complete [Bojańczyk et al. 2015]. The argument amounts to showing that for linear child-only programs it is enough to consider counter-examples of bounded *clique-width* [Courcelle and Olariu 2000]—a complexity measure of structures, similar to tree-width—and then evaluating programs over the usual clique decompositions, which are finitely labelled trees.

Organization. In Section 2 we introduce datalog and RTPQs, and explore the connections between them. Then, after a brief glance at containment over words and ranked trees in Section 3, we move on to unranked trees to discuss our main results in Section 4. All undecidability arguments are moved to Section 5 and all hardness proofs are presented in Section 6. We conclude in Section 7.

Note. A 12-pages long, preliminary version of this paper appeared in conference proceedings [Mazowiecki et al. 2014]. There, we proved that containment of child-only programs over words is in EXPSPACE in general and PSPACE-complete for linear programs; and over ranked trees it is in 3-EXPTIME in general and 2-EXPTIME-complete for linear programs. This left an exponential complexity gap for non-linear programs in both cases. Here, we close this gap using entirely different arguments: for arbitrary child-only programs we show PSPACE-completeness over words (Proposition 3.1) and 2-EXPTIME-completeness over ranked trees (Theorem 3.3). Moreover, the present paper gives full proofs of the results on unranked trees (Proposition 4.1, Theorem 4.2, and Theorem 4.6); in [Mazowiecki et al. 2014] we only gave very high level sketches for Theorem 4.2, and Theorem 4.6, omitting all arguments and automata constructions, and the reduction establishing Proposition 4.1 was skipped entirely.

2. DATALOG AND RTPQS

Both formalisms work over finite unordered unranked trees labeled with letters from a fixed infinite alphabet Σ. We write $nodes_t$ for the set of nodes of tree t, and $lab_t : nodes_t \to \Sigma$ for the function assigning labels to nodes. We use standard notation for axes: $\downarrow, \downarrow^+, \uparrow, \uparrow^+$ stand for child, descendant, parent, and ancestor relations. Binary relation $\sim$ holds between nodes with identical labels. We also have a unary predicate a for each $a \in \Sigma$, holding for the nodes labeled with a.

Many papers on tree-structured data work with a slightly different data model, called *data trees*, where each node has a label from a finite alphabet and a data value from an infinite data domain. In that model labels can be used explicitly in the formulas, but cannot be directly tested for equality, and data values can be tested for

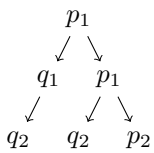
equality, but cannot be used explicitly (as constants). These two models are very similar, but not directly comparable for query languages with limited negation. However, we can easily incorporate additional finite alphabet to our setting, obtaining a generalization of the two settings, and the complexity results do not change. Forbidding the use of constants from the infinite alphabet may affect some of our lower bounds, but not the ones showing undecidability.

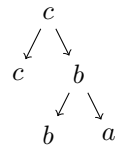
2.1. Datalog

We begin with a brief description of the syntax and semantics of datalog; for more details see [Abiteboul et al. 1995] or [Ceri et al. 1990]. A **datalog program** $\mathcal{P}$ over a relational signature S is a set of rules of the form $head \leftarrow body$, where $head$ is an atom over S and $body$ is a (possibly empty) conjunction of atoms over S written as a comma-separated list. There is a designated rule called the **goal** of the program. All variables in the body that are not used in the head, are implicitly quantified existentially. The size of a rule is the number of different variables that appear in it. The relational symbols, or predicates, in S fall into two categories. **Extensional** predicates are the ones explicitly stored in the database; they are never used in the heads of rules. In our setting they come from $\{\downarrow, \downarrow^+, \sim\} \cup \Sigma$. Alphabet Σ is infinite, but program $\mathcal{P}$ uses only its finite subset $\Sigma_{\mathcal{P}}$. **Intensional** predicates, used in the heads, are defined by the rules. The program is evaluated by generating all the atoms (over intensional predicates) that can be inferred from the underlying structure (tree) by applying the rules repeatedly, to the point of saturation, and then taking atoms matching the head of the goal rule. We write $\mathcal{P}(t)$ for the set of atoms returned by program $\mathcal{P}$ over tree t. Each inferred atom can be witnessed by a **proof tree**: an atom inferred by rule r from intensional atoms $A_1, A_2, \ldots, A_n$ is witnessed by a proof tree whose root has label r, and its children are the roots of proof trees for atoms A_i (if r has no intensional predicates in its body, the root has no children).

Example 2.1. The program below computes nodes from which one can reach label a along a path such that each node on the path has a child with identical label and a descendant with label b (or has label b itself):

$$\underline{P(X)} \leftarrow X{\downarrow}Y, P(Y), X{\downarrow}Y', X \sim Y', Q(X) \quad (p_1)$$
$$P(X) \leftarrow a(X) \quad (p_2)$$
$$Q(X) \leftarrow X{\downarrow}Y, Q(Y) \quad (q_1)$$
$$Q(X) \leftarrow b(X) \quad (q_2)$$





The intensional predicates are P and Q, and the goal is P. The proof tree shown in the center witnesses that P holds in the root of the tree on the right.

In this paper we consider only **monadic** programs, that is, programs whose intensional predicates are at most unary. Moreover, throughout the paper we assume that programs do not use 0-ary intensional predicates. For general programs this is merely for the sake of simplicity: one can always turn 0-ary predicate Q to unary predicate $Q(X)$ by introducing a dummy variable X. For connected and downward programs (described below) this restriction matters.

A datalog program is **linear**, if the right-hand side of each rule contains at most one atom with an intensional predicate (proof trees for such programs are single branches). For a datalog rule r, let G_r be the graph whose vertices are the variables used in r and edge is placed between X and Y if the body of r contains an atomic formula $X{\downarrow}Y$ or $X{\downarrow}^+Y$. A program $\mathcal{P}$ is **connected** if for each rule $r \in \mathcal{P}$, G_r is connected. We say that $\mathcal{P}$ is **downward** if for each rule $r \in \mathcal{P}$, G_r is a directed tree whose root is the variable

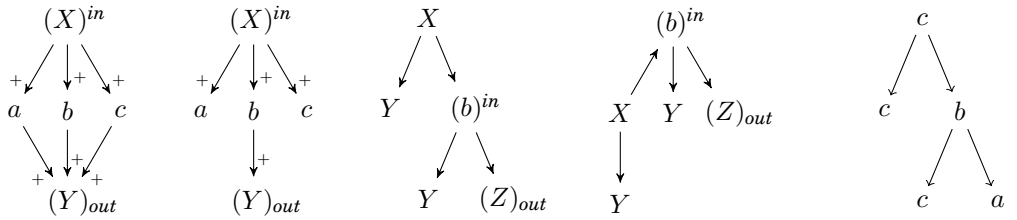


Fig. 1: Four graph patterns; all except the first one match in the tree on the right.

used in the head of r. The program from Example 2.1 is connected and downward, but not linear. In fact, each downward program is connected.

We work exclusively with connected programs. As argued in the introduction, this is a natural and common assumption. We write $\mathsf{Datalog}(\downarrow, \downarrow^+)$ for the class of *connected monadic* datalog programs, and $\mathsf{Datalog}(\downarrow)$ for *connected monadic* programs that do not use the relation $\downarrow^+$. For linear or downward programs we use combinations of letters L and D, e.g., $\mathsf{LD\text{-}Datalog}(\downarrow)$ means linear downward programs from $\mathsf{Datalog}(\downarrow)$.

Recall that **conjunctive queries** (CQs) are existential first order formulas of the form $\exists x_1 \dots \exists x_k\, \varphi$, where φ is a conjunction of atoms. Sometimes we also speak of (finite) **unions of conjunctive queries** (UCQs), corresponding to programs with a single intensional predicate (goal), which is never used in the bodies of rules. Like for datalog we use notation $\mathsf{CQ}(\downarrow, \downarrow^+)$, $\mathsf{CQ}(\downarrow)$, $\mathsf{UCQ}(\downarrow, \downarrow^+)$, $\mathsf{UCQ}(\downarrow)$.

Throughout the paper we work with the Boolean variant of containment, which is unsatisfiability of $\mathcal{P} \wedge \neg\mathcal{Q}$, where $\mathcal{P}$ and $\mathcal{Q}$ are treated as Boolean queries; that is, we ask if for each tree t, $\mathcal{P}(t) \neq \emptyset$ implies $\mathcal{Q}(t) \neq \emptyset$. In the unary variant of containment one asks if for each tree t, $\mathcal{P}(t) \subseteq \mathcal{Q}(t)$. Using well known methods one can reduce the unary variant to the Boolean one (not by simply rewriting queries, though), but the opposite reduction requires descendant or recursion. Hence, all our upper bounds transfer to unary containment, but our lower bounds transfer only for fragments allowing recursion or descendant.

2.2. RTPQs

We now move to the second formalism. Recall that Σ denotes the fixed infinite alphabet of node labels. Let Δ be an infinite set of variables and let $\tau \subseteq \{\downarrow, \downarrow^+, \uparrow, \uparrow^+\}$ be a nonempty set of axes (in [Abiteboul et al. 2011] only $\downarrow$ and $\downarrow^+$ are used). A **graph pattern** over τ is a directed multigraph with vertices labeled with elements of $\Sigma \cup \Delta$ and edges labeled with elements of τ. We write $\mathit{vertices}_\pi$ for the set of vertices of pattern π and $\mathit{lab}_\pi : \mathit{vertices}_\pi \to \Sigma \cup \Delta$ for the labeling function. Additionally, each pattern π has two distinguished nodes: $\mathit{in}(\pi)$ and $\mathit{out}(\pi)$.

Definition 2.2. A **homomorphism** $h : \pi \to t$ from a pattern π over τ to a tree t is a function $h : \mathit{vertices}_\pi \to \mathit{nodes}_t$ such that for all vertices v, w in π
— if $\mathit{lab}_\pi(v) \in \Sigma$, then $\mathit{lab}_t(h(v)) = \mathit{lab}_\pi(v)$;
— if $\mathit{lab}_\pi(v) = \mathit{lab}_\pi(w) \in \Delta$, then $h(v) \sim h(w)$; and
— h preserves binary relations from τ.

We say that π matches in t, written as $t \models \pi$, if there exists a homomorphism $h\colon \pi \to t$. We write $t, v \models \pi$ if h maps $\mathit{in}(\pi)$ to v and $t, v, w \models \pi$ if h also maps $\mathit{out}(\pi)$ to w.

Four examples of graphs patterns are given in Figure 1. Only the last three can be matched in the tree on the right: to match the first one we need letters a, b, c on a single branch. The last two patterns are equivalent, but they use different sets of relations: the third uses only $\downarrow$ and the fourth uses $\downarrow$ and $\uparrow$.

A **tree pattern query** (TPQ) over τ is a graph pattern π over τ that is a directed tree, whose *in* node is the root. For instance, the second and the fourth pattern in Figure 1 are TPQs. The third one is not a TPQ, because its *in* node is not in the root, but it can be rewritten into an equivalent TPQ (the fourth pattern) at the cost of using relation $\uparrow$. Finally, the first pattern is not a TPQ and is not equivalent to a TPQ, but it is equivalent to a union of 6 TPQs. The set of all TPQs over τ is denoted by $\mathsf{TPQ}(\tau)$. Let $\Pi = \pi_1 \cdot ... \cdot \pi_n$ be a word over alphabet $\mathsf{TPQ}(\tau)$, with $\pi_i \in \mathsf{TPQ}(\tau)$. A homomorphism $h : \Pi \to t$ is a sequence of homomorphisms $h_i : \pi_i \to t$ for $i = 1, \ldots, n$ such that $h_i(\mathit{out}(\pi_i)) = h_{i+1}(\mathit{in}(\pi_{i+1}))$ for all $i < n$. We write $t \models \Pi$ if there is a homomorphism $h : \Pi \to t$; $t, v \models \Pi$ if h maps $\mathit{in}(\pi_1)$ to v; and $t, v, w \models \Pi$ if h also maps $\mathit{out}(\pi_n)$ to w.

Definition 2.3 (*[Abiteboul et al. 2011]*). A **regular tree pattern query** (RTPQ) φ over τ is a regular expression over the alphabet $\mathsf{TPQ}(\tau)$. $L(\varphi)$ denotes the language generated by φ. We write $t, v, w \models \varphi$ iff there is $\Pi \in L(\varphi)$ such that $t, v, w \models \Pi$, and similarly for $t, v \models \varphi$ and $t \models \varphi$. The set of all RTPQs over τ is denoted by $\mathsf{RTPQ}(\tau)$.

Example 2.4. Let π be the fourth pattern in Figure 1. Then

$$\big[(a)^{\mathit{in}} \downarrow (X)_{\mathit{out}}\big] \cdot \pi^* \cdot \big[(a)^{\mathit{in}}_{\mathit{out}}\big]$$

is an RTPQ that selects pairs of a-labelled nodes that are connected by a path (going down the tree) such that each internal node on the path has label b and has a child with the same label as one of its siblings (or itself).

Example 2.5. To illustrate further the power of RTPQs, we show how to simulate the n-bit binary counter, enumerating values from 0 to $2^n - 1$, with an RTPQ of size $\mathcal{O}(n^2)$. To increase the counter, we need to find the least significant 0, change it to 1 and change all less significant bits to 0; all more significant bits remain unchanged. Skipping $\downarrow$ to ease the notation, we can express this with patterns

$$\begin{aligned} \mathit{inc}_i &= (X_1)^{\mathit{in}} X_2 \cdots X_{n-i} 01 \cdots 1 (X_1)_{\mathit{out}} X_2 \cdots X_{n-i} 10 \cdots 0 \quad \text{for } \ i < n\,, \\ \mathit{inc}_n &= (0)^{\mathit{in}} 1 \cdots 1 (1)_{\mathit{out}} 0 \cdots 0\,. \end{aligned}$$

Then, we combine patterns inc_i into a counter going from 0 to $2^n - 1$:

$$c_n = \mathit{val}_0 \cdot \big(\mathit{inc}_1 \cup \mathit{inc}_2 \cup \cdots \cup \mathit{inc}_n\big)^* \cdot \mathit{val}_{2^n - 1}$$

where val_k is number k stored in n bits in binary, e.g., $\mathit{val}_0 = (0)^{\mathit{in}}_{\mathit{out}} 0 \cdots 0$. This expression works if we run it on a tree consisting of a single branch: if such tree satisfies c_n, then it must contain all values from 0 to $2^n - 1$. Note that by locating the *out* node in pattern inc_i on the $(n+1)$-th position, we pass many values between two consecutive TPQs, because the following nodes must overlap with the initial n nodes of the next TPQ. On arbitrary trees this is no longer the case, but we can make the expression work by running it up the tree, i.e., replacing $\downarrow$ with $\uparrow$.

2.3. Equiexpressiveness and witnessing patterns

The usual correspondence between patterns and CQs holds also for trees over infinite alphabet, except that relation $\sim$ in patterns is not represented explicitly, but by repeated labels from Δ. Hence, the simplest translation from patterns to CQs involves quadratic blowup (it can be avoided). As observed in [Gottlob et al. 2006], each satisfiable graph pattern can be expressed as a union of TPQs reflecting different ways of mapping the pattern to a tree. While the size of the TPQs can be bounded by the size of the pattern, their number is exponential. Given this, the following is routine.

PROPOSITION 2.6. *The following classes of queries have the same expressive power:*
(1) RTPQ$(\downarrow, \downarrow^+, \uparrow, \uparrow^+)$ *and* L-Datalog$(\downarrow, \downarrow^+)$*;* *(3)* RTPQ$(\downarrow, \uparrow)$ *and* L-Datalog$(\downarrow)$*;*
(2) RTPQ$(\downarrow, \downarrow^+)$ *and* LD-Datalog$(\downarrow, \downarrow^+)$*;* *(4)* RTPQ$(\downarrow)$ *and* LD-Datalog$(\downarrow)$.
Translations to datalog are polynomial; translations to RTPQs are exponential.

PROOF. We show that for every linear monadic datalog program $\mathcal{P}$, there is an equivalent RTPQ $\varphi_{\mathcal{P}}$, and conversely, for every RTPQ φ there is an equivalent linear monadic program $\mathcal{P}_\varphi$. The remaining claims follow easily from these two general translations. Recall that we assume throughout the paper that all programs are connected. In the third and fourth claim this is necessary, because disconnected graph patterns cannot be expressed as unions of TPQs without the long axes, $\downarrow^+$, $\uparrow^+$. But if programs can use long axes, like in the first claim, they can easily move to any node in the tree and this assumption is not necessary any more.

From RTPQs to datalog. If $\varphi = \pi$, an equivalent datalog program is of the form $G(X) \leftarrow body_\pi$, where each node of π becomes a variable in the program, the *in* node becomes X, and appropriate relations are copied. For example, TPQ π from Example 2.4 gives $G(X) \leftarrow X{\downarrow}Y, X{\downarrow}Z, X'{\downarrow}X, X'{\downarrow}Y', b(X), Y \sim Y'$. Note how the repeated variable Y in π is reflected by the atomic formula $Y \sim Y'$.

In general, let $\mathcal{A}_\varphi$ be a finite automaton over the alphabet consisting of TPQs occurring in φ that recognizes $L(\varphi)$. The program $\mathcal{P}_\varphi$ is created as follows. For each state q of $\mathcal{A}_\varphi$, we introduce a new intensional predicate $q(X)$, and for each transition (q, π, q') in $\mathcal{A}_\varphi$ we add a rule $q(X) \leftarrow body_\pi, q'(Y)$, where Y is the variable corresponding to the *out* node of π ($Y = X$ if the *in* and *out* nodes coincide). If q' is accepting, we also add rule $q(X) \leftarrow body_\pi$. The goal of $\mathcal{P}_\varphi$ is $G(X) \leftarrow q_0(X)$, where q_0 is the initial state of $\mathcal{A}_\varphi$. For instance, an automaton corresponding to the RTPQ $\left[(a)^{in} \downarrow (X)_{out}\right] \cdot \pi^* \cdot \left[(a)^{in}_{out}\right]$ from Example 2.4 has states p, q, r, where p is the initial state and r is the only accepting state, with transitions $(p, [(a)^{in} \downarrow (X)_{out}], q)$, (q, π, q), and $(q, [(a)^{in}_{out}], r)$. This gives the following program

$$
\begin{aligned}
&G(X) \leftarrow P(X)\,,\\
&P(X) \leftarrow a(X), X{\downarrow}Y, Q(Y)\,,\\
&Q(X) \leftarrow X{\downarrow}Y, X{\downarrow}Z, X'{\downarrow}X, X'{\downarrow}Y', b(X), Y \sim Y', Q(Z)\,,\\
&Q(X) \leftarrow a(X), R(X)\,,\\
&Q(X) \leftarrow a(X)\,;
\end{aligned}
$$

since there are no transitions from state r, the penultimate rule can be dropped.

From datalog to RTPQs. Let $\mathcal{P}$ be a linear monadic datalog program. Without loss of generality we can assume that each rule of $\mathcal{P}$ is consistent, i.e., its body is satisfiable in a tree. Turn the conjunction of all the atoms in the body over extensional predicates into a graph pattern. The *in* node is the node corresponding to the variable in the head of the rule and the *out* node is the node corresponding to the variable used by the only intensional predicate in the body of the rule (here we use the linearity of $\mathcal{P}$). If there is no intensional predicate in the rule's body, then the *out* node can be any node as this pattern will not be connected to any further patterns; set *out* $=$ *in* for concreteness. As we have mentioned, this graph pattern (seen as a binary query) can be rewritten as a union of exponentially many TPQs. Let $\mathcal{T}$ be the set of all TPQs obtained by transforming $\mathcal{P}$'s bodies.

We now construct an automaton $\mathcal{A}$ reading words over $\mathcal{T}$. The states of $\mathcal{A}$ are the intensional predicates of $\mathcal{P}$, plus the accepting state $\top$. The transitions of $\mathcal{A}$ follow the rules of $\mathcal{P}$. Let π be a TPQ obtained from rule $R(X) \leftarrow \ldots, Q(Y)$, where Q is an intensional predicate. We add transition (R, π, Q) to $\mathcal{A}$. For rules without intensional predicates in the body, we use $\top$ as the target state of the transition. The initial state

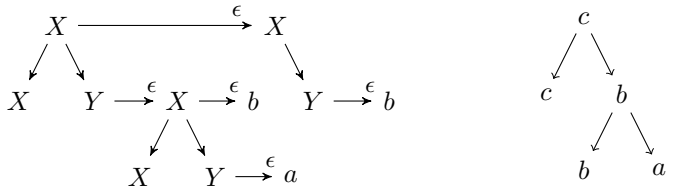


Fig. 2: A pattern with ϵ-edges and a tree where it can be matched.

is the goal of $\mathcal{P}$. Next, we turn the automaton $\mathcal{A}$ into an equivalent regular expression $\varphi_{\mathcal{P}}$ (of exponential size). Expression $\varphi_{\mathcal{P}}$ is an RTPQ and (seen as a unary query) it is equivalent to $\mathcal{P}$.

Rewriting graph patterns into unions of tree patterns gives exponential alphabet $\mathcal{T}$, and expression $\varphi_{\mathcal{P}}$ is exponential in the size of automaton $\mathcal{A}$'s state-space (which is linear in the size of $\mathcal{P}$). Thus, expression $\varphi_{\mathcal{P}}$ is single exponential in the size of program $\mathcal{P}$. Since exponential blow-up is unavoidable when translating automata to regular expressions, it is also unavoidable here. □

When rules are translated into patterns, and *vice versa*, variables correspond to nodes; the variable in the head corresponds to the *in* node, and the variable used in the intensional atom (unique in linear programs) corresponds to the *out* node. Thus, we may speak of *in* or *out* variables, and head or intensional nodes.

The connection between datalog and patterns goes beyond linear programs. Consider patterns with an additional relation ϵ, interpreted as node equality; we indicate the use of ϵ-edges by including ϵ in the signature, e.g., $\mathsf{TPQ}(\downarrow, \downarrow^+, \epsilon)$. A homomorphism from such a pattern π is a family of homomorphisms from all maximal ϵ-free subpatterns of π, such that nodes connected by ϵ-edges are mapped to the same tree node. Note that pattern nodes labelled with the same variable must be mapped to tree nodes with the same label only if they are in the same ϵ-free subpattern. For instance, in Figure 2 the pattern on the left can be matched in the tree on the right by mapping the nodes in the top layer to the root of the tree, the first node of the middle layer to the left child of the root, the remaining nodes of the middle layer to the right child of the root, the first node of the bottom layer to the b-leaf, and the remaining nodes of the bottom layer to the a-leaf (the ϵ-edges are horizontal only for readability).

Since proof trees of linear programs are single branches, they can be interpreted as words of graph patterns (corresponding to rule bodies). Each such word has a natural representation as a pattern with ϵ-edges, obtained by inserting an ϵ-edge from the *out* node of each pattern to the *in* node of the following pattern. This gives a pattern representation for proof trees of linear programs. To generalize it to non-linear programs, let π_r be the graph pattern corresponding to the conjunction of extensional atoms in the body of rule r, and let $\mathcal{G}_{\mathcal{P}}$ be the set of all π_r's. Then, with each proof tree one can associate a graph pattern by replacing each r-labeled node by the pattern π_r together with appropriate ϵ-edges between π_r's intensional nodes and its children's head nodes. Patterns obtained this way are called **witnessing patterns**. Note that each maximal connected ϵ-free subpattern of a witnessing pattern corresponds to a rule of $\mathcal{P}$. Figure 2 shows the witnessing pattern for the proof tree from Example 2.1.

2.4. Automata

Throughout the paper all decidability results use automata constructions. We briefly recall the standard automata model for ranked trees here.

A (bottom-up) **tree automaton** $\mathcal{A} = \langle \Gamma, Q, \delta, F \rangle$ on at most R-ary trees consists of a finite alphabet Γ, a finite set of states Q, a set of accepting states $F \subseteq Q$, and transition relation $\delta \subseteq \bigcup_{k=0}^{R} Q \times \Gamma \times Q^k$. A run on a tree t over Γ is a labeling ρ of t with elements of Q consistent with the transition relation, i.e., if v has children $v_1, v_2, \ldots, v_k$ with $k \leq R$, then $(\rho(v), lab_t(v), \rho(v_1), \ldots, \rho(v_k)) \in \delta$. In particular, if v is a leaf we have $(q, a) \in \delta$. Run ρ is accepting if it assigns a state from F to the root. A tree is accepted by $\mathcal{A}$ if it admits an accepting run. The language recognized by $\mathcal{A}$, denoted by $L(\mathcal{A})$, is the set of all accepted trees. We recall that testing emptiness of a tree automaton can be done in PTIME, but complementation involves an exponential blow-up.

To work on ordered unranked trees we present them as binary trees, using the usual "previous sibling, last child" encoding. That is, an unranked tree t is turned into a binary tree t_b as follows: copy the root from t to t_b; for each node v from t let v_b be its copy in t_b, as the left child of v_b we put a copy of the previous sibling of v, and as the right child we put a copy of the last child of v. If v is a left-most child, we put a stub node labelled with a special symbol $\# \notin \Sigma$ as the left child of v_b; similarly, if v is a leaf we put a stub node as the right child of v_b. Thus we can use the same automata for the encoded ordered unranked trees. To use them for unordered unranked trees, just pick an arbitrary order on siblings.

As an intermediate automata model, closer to datalog than the bottom-up automata, we shall use the two-way alternating automata introduced in [Cosmadakis et al. 1988]. A **two-way alternating automaton** $\mathcal{A} = \langle \Gamma, Q, q_I, \delta \rangle$ consists of an alphabet Γ, a finite set of states Q, an initial state $q_I \in Q$, and a transition function

$$\delta \colon Q \times \Gamma \to \mathrm{BC}^+\big(Q \times \{-1, 0, 1\}\big)$$

describing actions of automaton $\mathcal{A}$ in state q in a node with label a as a positive Boolean combination of atomic actions of the form $(p, d) \in Q \times \{-1, 0, 1\}$.

A run ρ of $\mathcal{A}$ over tree t is a tree labelled with pairs (q, v), where q is a state of $\mathcal{A}$ and v is a node of t, whose root is labelled with the pair consisting of q_0 and the root of t, and if a node of ρ with label (q, v) has children with labels $(q_1, v_1), \ldots, (q_n, v_n)$, and v has label a in t, then there exist $d_1, \ldots, d_n \in \{-1, 0, 1\}$ such that

— v_i is a child of v in t for all i such that $d_i = 1$;
— $v_i = v$ for all i such that $d_i = 0$;
— v_i is the parent of v in t for all i such that $d_i = -1$; and
— Boolean combination $\delta(q, a)$ evaluates to *true* when atomic actions $(q_1, d_1), \ldots, (q_n, d_n)$ are substituted by *true* and the remaining ones by *false*.

Tree t is accepted by automaton $\mathcal{A}$ if it admits a finite run. By $L(\mathcal{A})$ we denote the language recognized by $\mathcal{A}$; that is, the set of trees accepted by $\mathcal{A}$.

Example 2.7. Consider the language of trees over the alphabet $\{a_1, a_2, \ldots, a_n\}$, in which some branch contains all letters. To recognize this language, a two-way alternating automaton nondeterministically chooses a path down the tree, and at some point decides to stop and send up the tree n subprocedures, each looking for one of the letters a_i: the states are $q_0, q_1, \ldots, q_n$, the initial state is q_0, and the transition relation is

$$\begin{aligned} \delta(q_0, a_i) &= (q_0, 1) \vee \big((q_1, 0) \wedge (q_2, 0) \wedge \cdots \wedge (q_n, 0)\big)\,, \\ \delta(q_i, a_j) &= (q_i, -1)\,, \\ \delta(q_i, a_i) &= \mathrm{true} \end{aligned}$$

for $i = 1, 2, \ldots, n$ and $j \neq i$. Note that each equivalent (nondeterministic) bottom-up automaton has exponentially many states.

According to the definition above, two-way alternating automata only distinguish between going up, down, and staying where they are. In a more general model, appropriate for ordered ranked trees, one could also distinguish between going to the first child, the second child, etc. Given that our datalog programs are not able to make such distinction, this simplified definition suffices.

The notion of two-way alternating automata is naturally adapted to words by viewing words as unary trees.

The computation model of two-way alternating automata is very similar to that of datalog programs, making them a perfect intermediate formalism on the road to nondeterministic bottom-up automata. From there one continues thanks to the following.

PROPOSITION 2.8 ([COSMADAKIS ET AL. 1988; VARDI 1998]). *For each two-way alternating automaton $\mathcal{A}$ (over words or ranked trees), there exist single-exponential nondeterministic bottom-up automata recognizing the language $L(\mathcal{A})$ and its complement, whose states and transitions can be enumerated in polynomial working memory.*

Notice that complementing two-way alternating automata is not trivial because there can be infinite runs that are not accepting.

3. RANKED TREES

To warm up, we first look at the special case of unary trees—or simply words—where complexities are different than for trees of higher arity, and where some ideas can be illustrated without much of the technical difficulty of trees of higher arity. Following the word interpretation, the relations $\downarrow$ and $\downarrow^+$ should be understood as *next position* and *following position*.

3.1. Special case: words

Over data words and data trees, containment of monadic datalog programs with descendant is undecidable—even containment of linear monadic programs in unions of conjunctive queries [Abiteboul et al. 2013, Proposition 3.3]. As discussed in Section 2, despite obvious similarities between our setting and the setting of data trees, neither lower bounds nor upper bounds carry over immediately. The reduction in [Abiteboul et al. 2013] relies on the presence of finite alphabet and cannot be directly adapted to our setting, but with a little effort the use of finite alphabet can be eliminated. Once descendant is disallowed, we immediately regain decidability. This may seem a very strong restriction, but in fact—as explained in the introduction—child-only programs can simulate the descendant relation with recursion as long as equality tests are local.

PROPOSITION 3.1. *Containment over words is*
(*1*) *undecidable for* LD-Datalog$(\downarrow, \downarrow^+)$ *(even containment in UCQs);*
(*2*) PSPACE*-complete for* Datalog$(\downarrow)$.

The proof of item (1) is moved to Section 5 (Proposition 5.1). It is a relatively direct reduction from the standard grid tiling problem. We encode the grid as a word built from its consecutive rows. We rely crucially on the fact that in words (as opposed to trees) positions have unique successors. The programs use $\downarrow$ to verify the horizontal relation, and $\downarrow^+$ and label equalities to verify the vertical relation. For the lower bound in item (2) we show an easy reduction based on the binary counter encoding from Section 2. We prove that even satisfiability for L-Datalog$(\downarrow)$ programs is PSPACE-hard (see Section 6, Proposition 6.1).

In this section we prove that containment for Datalog$(\downarrow)$ is in PSPACE. We begin by showing that it suffices to check containment over a finite alphabet. Recall that for a program $\mathcal{R}$ we write $\Sigma_{\mathcal{R}}$ for the set of labels explicitly used in $\mathcal{R}$.

LEMMA 3.2. *Let $\mathcal{P}, \mathcal{Q} \in \mathsf{Datalog}(\downarrow)$ over an infinite alphabet Σ, and let $\Sigma_0 = \Sigma_{\mathcal{P}} \cup \{\bot_1, \bot_2, \ldots, \bot_n\}$, where $\bot_1, \bot_2, \ldots, \bot_n \in \Sigma \setminus (\Sigma_{\mathcal{P}} \cup \Sigma_{\mathcal{Q}})$ are distinct fresh labels and n is the maximal size of a rule in $\mathcal{P}$ or $\mathcal{Q}$. Then, $\mathcal{P} \wedge \neg\mathcal{Q}$ is satisfiable in a word over Σ if and only if it is satisfiable in a word over Σ_0.*

PROOF. The right to left implication is obvious. Arguing in the other direction, suppose we have a word v over Σ that satisfies $\mathcal{P} \wedge \neg\mathcal{Q}$. Our goal is to change the word v into a word over Σ_0. Without loss of generality, we can assume that v does not use labels from $\Sigma_{\mathcal{Q}} \setminus \Sigma_{\mathcal{P}}$. The procedure processes consecutive letters of the word v and puts some of them on a list ℓ, initially empty. We denote the size of the list by $|\ell|$. Let i be the position of the next letter to process. If the letter v_i is from the alphabet $\Sigma_{\mathcal{P}}$ we do nothing. If v_i is already on the list, we move it to the beginning of the list and proceed to the next letter. Assume $v_i \notin \Sigma_{\mathcal{P}}$ and $v_i \notin \ell$. If $|\ell| < n$, we add v_i at the beginning of ℓ and proceed to the next letter. If $|\ell| = n$, we modify the word v as follows:

(1) let a be the last letter in ℓ;
(2) let b be a fresh letter that does not occur in v;
(3) for all $j > i$ such that $v_j = a$, change v_j to b;
(4) for all $j \geq i$ such that $v_j = v_i$, change v_j to a; and
(5) move the letter a to the beginning of ℓ.

After processing the last letter of v we obtain a word using only letters from $\Sigma_{\mathcal{P}}$ and at most n letters from $\Sigma \setminus (\Sigma_{\mathcal{P}} \cup \Sigma_{\mathcal{Q}})$. Hence, we can turn it into a word over Σ_0, without changing the equalities among letters of the word and letters from $\Sigma_{\mathcal{P}} \cup \Sigma_{\mathcal{Q}}$. Let us denote this word by u.

For instance, suppose that $n = 3$ and $v = a_1a_1ba_2a_2ba_3a_3ba_4a_4b$, where $b \in \Sigma_{\mathcal{P}}, a_i \notin \Sigma_{\mathcal{P}}$ and a_i are all different. Then the modified word is $u = a_1a_1ba_2a_2ba_1a_1ba_2a_2b$. It is clear that the whole procedure never changes equality of labels within distance n, and does not affect labels from $\Sigma_{\mathcal{P}} \cup \Sigma_{\mathcal{Q}}$. Since within the rules of $\mathcal{P}, \mathcal{Q}$ all variables are connected with $\downarrow$, it follows that u satisfies $\mathcal{P} \wedge \neg\mathcal{Q}$. □

By the results of [Gottlob and Koch 2004], monadic datalog programs over ranked and unranked trees (hence, also words) over a finite alphabet can be translated to MSO. Hence, the above lemma immediately gives decidability of the containment problem. To get the PSPACE upper bound we shall translate programs to tree automata without passing through MSO. As an intermediate formalism we shall use two-way alternating automata.

PROOF OF PROPOSITION 3.1. It suffices to show that for a given program $\mathcal{R} \in \mathsf{Datalog}(\downarrow)$ and finite alphabet Σ_0 one can construct in PTIME a two-way alternating automaton $\mathcal{A}_{\mathcal{R}}$ recognizing words over Σ_0 that satisfy $\mathcal{R}$. Indeed, by Lemma 3.2 it suffices to check satisfiability of $\mathcal{P} \wedge \neg\mathcal{Q}$ in words over the linear-size alphabet Σ_0. From automata $\mathcal{A}_{\mathcal{P}}$ and $\mathcal{A}_{\mathcal{Q}}$ by Proposition 2.8 we obtain one-way non-deterministic automata $\mathcal{B}_{\mathcal{P}}, \mathcal{B}_{\neg\mathcal{Q}}$ of exponential size that recognize respectively the languages $L(\mathcal{A}_{\mathcal{P}})$ and the complement of $L(\mathcal{A}_{\mathcal{Q}})$. From this we easily get a product automaton $\mathcal{B}_{\mathcal{P}\wedge\neg\mathcal{Q}}$ equivalent to the query $\mathcal{P} \wedge \neg\mathcal{Q}$. The size of $\mathcal{B}_{\mathcal{P}\wedge\neg\mathcal{Q}}$ is exponential in size of $\mathcal{P}, Q$, but its states and transitions can be generated on the fly in polynomial space. Indeed, Proposition 2.8 guarantees that the states of $\mathcal{B}_{\mathcal{P}}$ and $\mathcal{B}_{\neg\mathcal{Q}}$ have polynomial representation and their transition relations can be enumerated in polynomial space, so each time we need a transition in the product automaton from a given state over a given letter, we enumerate the transition relations of $\mathcal{B}_{\mathcal{P}}$ and $\mathcal{B}_{\neg\mathcal{Q}}$ discarding the obtained tuples until we find the ones we are looking for. To check emptiness of $\mathcal{B}_{\mathcal{P}\wedge\neg\mathcal{Q}}$ we make a simple nondeterministic reachability test. By Savitch's theorem, this gives a PSPACE algorithm.

Let us fix a program $\mathcal{R} \in \mathsf{Datalog}(\downarrow)$ and a finite alphabet Σ_0. Since we work over words and consider child-only connected programs, without loss of generality we can assume that each rule r is of the form

$$H(X_0) \leftarrow \bigwedge_{i=k}^{\ell-1} X_i \downarrow X_{i+1},\ \psi(X_k, X_{k+1}, \ldots, X_\ell)$$

where $k \leq 0 \leq \ell$ (notice that k might be negative) and $\psi(X_k, X_{k+1}, \ldots, X_\ell)$ is a conjunction of atoms over unary predicates and $\sim$; that is, it does not use $\downarrow$. This means that the pattern π_r corresponding to the body of r is a word.

In the automaton $\mathcal{A}_\mathcal{R} = \langle \Sigma_0, Q, q_0, \delta \rangle$ we are about to define, we allow transitions of a slightly generalized form: the transition function δ assigns to each state-letter pair a positive Boolean combination of elements of

$$Q \times \{-N, -(N-1), \ldots, 0, \ldots, N-1, N\}$$

for a fixed constant $N \in \mathbb{N}$, rather than just $Q \times \{-1, 0, 1\}$. The semantics of this is the natural one: (q, k) means that the automaton moves by k positions (left or right depending on the sign of k) and changes state to q. Each generalized automaton can be transformed to a standard one at the cost of enlarging the state-space by the factor of $2N + 1$. In our case N will be bounded by the maximal number of variables used in any rule of program $\mathcal{R}$.

Let us describe the automaton $\mathcal{A}_\mathcal{R}$. The state-space Q is

$$\Sigma_0 \cup \mathcal{R} \cup \{q_0\}\,;$$

that is, it consists of the letters from Σ_0, the rules of program $\mathcal{R}$, and an additional initial state q_0. The transition relation δ is defined as follows. In the initial state q_0, regardless of the current letter, we loop moving to the right until we reach the position in the word where we start evaluating program $\mathcal{R}$:

$$\delta(q_0, \_) = (q_0, +1) \vee (r_{\text{goal}}, 0)\,,$$

where r_{goal} is the goal rule of $\mathcal{R}$. When we are in state $r \in \mathcal{R}$, regardless of the current letter, we check that the body of r can be matched in the input word in such a way that X_0 is mapped to the current position:

$$\delta(r, \_) = \bigwedge_{a(X_i) \in B_r} (a, i) \ \wedge \bigwedge_{X_i \sim X_j \in B_r} \bigvee_{b \in \Sigma_0} (b, i) \wedge (b, j) \ \wedge \bigwedge_{P(X_i) \in B_r} \bigvee_{r' \in \mathcal{R}_P} (r', i)\,,$$

where B_r is the set of atoms in the body of rule r and $\mathcal{R}_P \subseteq \mathcal{P}$ is the set of rules defining intensional predicate P. In state $a \in \Sigma_0$ we simply check that the letter in the current position is a:

$$\delta(a, a) = \textit{true}\,, \quad \text{and} \quad \delta(a, b) = \textit{false for } b \neq a\,.$$

Checking correctness and the size bounds for $\mathcal{A}_\mathcal{R}$ poses no difficulties. □

3.2. General case

The results for words can be lifted to ranked trees: complexities are higher, but the general picture remains the same.

THEOREM 3.3. *Over ranked trees containment is*
(*1*) *undecidable for* LD-Datalog$(\downarrow, \downarrow^+)$ *(even containment in UCQs);*
(*2*) *2-*EXPTIME*-complete for* Datalog$(\downarrow)$.

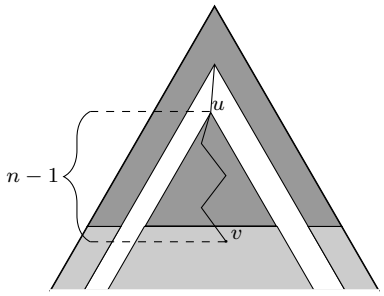


Fig. 3: Processing node v

Like for words, the undecidability proof is moved to Section 5 (Proposition 5.3); it is a slight modification of the undecidability proof in Proposition 3.1. The 2-EXPTIME lower bound can be obtained already for D-Datalog(↓) and L-Datalog(↓); both reductions are given in Section 6 (Proposition 6.3 and Proposition 6.4). The proof is a reduction from the emptiness problem for alternating Turing machines using at most 2^n tape cells. We encode runs of alternating Turing machines as trees with nodes labelled with the states of the respective configurations, and below each such node we attach a full binary tree of depth n, whose leaves store the contents of the tape cells. In this encoding the tape cells of consecutive configurations are within small distance (linear in n), which allows the programs to verify the correctness of the encoding.

In this section we show the 2-EXPTIME upper bound, and again we begin by bounding the size of the alphabet.

LEMMA 3.4. *Consider programs $\mathcal{P}, \mathcal{Q} \in$ Datalog(↓) and a number $R \in \mathbb{N}$, and let $\Sigma_0 = \Sigma_\mathcal{P} \cup \{\bot_1, \bot_2, \ldots, \bot_{R^n}\}$, where $\bot_1, \bot_2, \ldots, \bot_{R^n} \in \Sigma \setminus (\Sigma_\mathcal{P} \cup \Sigma_\mathcal{Q})$ are distinct fresh labels and n is the maximal size of a rule in $\mathcal{P}$ or $\mathcal{Q}$. Then, $\mathcal{P} \wedge \neg\mathcal{Q}$ is satisfiable in an at most R-ary tree over Σ if and only if is satisfiable in an at most R-ary tree over Σ_0.*

PROOF. Analogously to Lemma 3.2, we describe a procedure that traverses the tree with labels from $\Sigma_\mathcal{P} \cup (\Sigma \setminus \Sigma_\mathcal{Q})$ in a top-down fashion, level by level, and replaces labels from $\Sigma \setminus \Sigma_\mathcal{Q}$ with elements of $\{\bot_1, \bot_2, \ldots, \bot_{R^n}\}$. The set of processed nodes always consists of i full levels starting from the root, and some nodes from level $i+1$.

Let v be a node on level $i+1$ to process (see Fig. 3), and let u be the node $n-1$ edges up the tree (or the root if v is too close to the root). Suppose the label of v is a. If $a \in \Sigma_0$, we do nothing. Assume $a \notin \Sigma_0$. Pick a label $\bot_j$ that does not appear in processed descendants of u, nor in u itself. We can always find such a label because the number of processed descendants of u (including u itself) is bounded by $\sum_{i=0}^{n-1} R^i = \frac{R^n-1}{R-1} < R^n$, and so is the number of labels from $\{\bot_1, \bot_2, \ldots, \bot_{R^n}\}$ used in these nodes. Let $b \in \Sigma \setminus \Sigma_0$ be a fresh label. We now replace all appearances of $\bot_j$ with b, but only in unprocessed descendants of node u. Observe that these nodes are separated from nodes that keep their label $\bot_j$ by distance at least n. Next, we replace all appearances of a with $\bot_j$, but only in the unprocessed descendants of u. Again, the distance from these nodes to other nodes with label a or $\bot_j$ is at least n. Thus, the modification does not affect the outcome of any label comparison done by rules of programs in $\mathcal{P} \wedge \neg\mathcal{Q}$ (because they use only the short axis and are connected).

After all nodes are processed, all labels in t are from Σ_0 and t still satisfies $\mathcal{P} \wedge \neg\mathcal{Q}$. □

Like for words, by [Gottlob and Koch 2004] we can conclude decidability of containment. A more detailed complexity analysis in [Frochaux et al. 2014] shows that containment of Datalog(↓) programs without $\sim$ on trees over a finite alphabet is EXPTIME-

complete. We have already reduced containment over infinite alphabet to containment over a finite alphabet of exponential size. We shall now see that $\sim$ can be eliminated without further blow-up, and the 2-EXPTIME upper bound follows from the bound of [Frochaux et al. 2014]. For the sake of completeness, we include the full proof, rather than just invoking the result of [Frochaux et al. 2014]. It relies on the translation of programs to two-way alternating automata. As a side comment, [Frochaux et al. 2014] also gives a 2-EXPTIME upper bound for programs allowing descendant, for which we have undecidability over infinite alphabet; this means that in the presence of descendant the reduction to a finite alphabet fails.

PROOF OF THEOREM 3.3. For a given program $\mathcal{R} \in \mathsf{Datalog}(\downarrow)$ with rules of size at most n and a finite alphabet Σ_0, we shall construct a two-way alternating automaton $\mathcal{A}_{\mathcal{P}}$ of size $\mathcal{O}(\|\mathcal{R}\| \cdot |\Sigma_0|^n \cdot n)$ recognizing trees over Σ_0 that satisfy $\mathcal{R}$; here, $\|\mathcal{R}\|$ is the total size of program $\mathcal{R}$. The construction will take time polynomial in the size of the input and output. Using this construction, like for words, by Lemma 3.4 and Proposition 2.8 we reduce the containment problem to the emptiness problem for a nondeterministic tree automaton, this time of double exponential size, and test emptiness with the standard PTIME algorithm.

Let us fix a program $\mathcal{R} \in \mathsf{Datalog}(\downarrow)$ and a finite alphabet Σ_0. Given that we are only interested in trees over alphabet Σ_0, we can eliminate the use of $\sim$ from $\mathcal{R}$: if a rule contains $X \sim Y$ we replace this rule with $|\Sigma_0|$ variants in which $X \sim Y$ is replaced with $a(X), a(Y)$ for $a \in \Sigma_0$. The size of the program grows by a $\mathcal{O}(|\Sigma_0|^n)$ factor; the size of the rules grows only by a constant factor.

Since we are working on trees we can further transform the program so that the patterns corresponding to the rules of the program are trees (with *in* and *out* nodes positioned arbitrarily). Indeed, it can be done by unifying variables X and Y whenever the rule contains $X \downarrow Z$ and $Y \downarrow Z$ for some variable Z, and removing rules containing atom $U \downarrow U$, or atoms $a(U)$ and $b(U)$ for some variable U and distinct letters a and b. This modification does not increase the size of the program.

Finally, we rewrite each rule into a set of rules of the form

$$H(X_0) \leftarrow a(X_0) \wedge \psi_0(X_0) \wedge \bigwedge_{i=1}^{\ell} \mathrm{ax}_i(X_0, X_i) \wedge \psi_i(X_i)$$

where $a \in \Sigma_0$, $\mathrm{ax}_i(X_0, X_i)$ is either $X_0 \downarrow X_i$ or $X_i \downarrow X_0$, and $\psi_i(X_i)$ is a conjunction of (monadic) intensional atoms. That is, one rule can only test the label and some intensional predicates for the current node, and demand existence of neighbours (children or parents) satisfying some intensional predicates. This modification introduces auxiliary intensional predicates, but the size of the program increases only by $\mathcal{O}(n)$ factor.

The resulting program is essentially a two-way alternating automaton, only given in a different syntax. □

4. UNRANKED TREES

We have seen in Section 3 that over ranked trees containment is undecidable for downward programs, but decidable for child-only programs. Over unranked trees exactly the opposite happens. Containment for general programs, even for $\mathsf{L}\text{-}\mathsf{Datalog}(\downarrow, \downarrow^+)$, remains undecidable as explained in Section 3, but the reduction for $\mathsf{LD}\text{-}\mathsf{Datalog}(\downarrow, \downarrow^+)$ in Theorem 3.3 relies heavily on the fixed number of children and does not go through for unranked trees. In Section 4.1 we obtain decidability even for $\mathsf{D}\text{-}\mathsf{Datalog}(\downarrow, \downarrow^+)$. More surprisingly, we lose decidability for $\mathsf{Datalog}(\downarrow)$: using non-linearity and recursion we can simulate $\downarrow^+$ to a point sufficient to repeat the reduction for words (see Section 5).

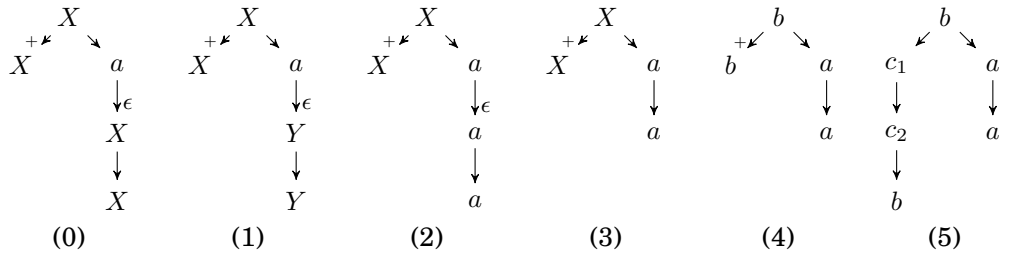


Fig. 4: Consecutive steps in transforming a pattern (0) into a canonical model (5).

PROPOSITION 4.1. *Over unranked trees containment of* Datalog($\downarrow$) *programs in* UCQ($\downarrow$) *queries is undecidable.*

In Section 4.2 we show that containment for *linear* Datalog($\downarrow$) programs is decidable.

4.1. Downward programs

The aim of this section is to prove the following theorem.

THEOREM 4.2. *Over unranked trees containment is 2-*EXPTIME*-complete for* D-Datalog($\downarrow, \downarrow^+$)*, and* EXPSPACE*-complete for* LD-Datalog($\downarrow, \downarrow^+$).

As usual, the proofs of the lower bounds are moved to Section 6 (Proposition 6.2 and Proposition 6.3). In the non-linear case, hardness does not require $\downarrow^+$. This is not so surprising: already in Proposition 4.1 we simulate $\downarrow^+$ using non-linearity. The 2-EXPTIME-hardness reduction is the same as for D-Datalog($\downarrow$) over ranked trees (Theorem 3.3). The EXPSPACE-hardness for linear downward programs is based on similar ideas, but due to technical difficulties this is the most involved reduction.

The proof of the positive claims of Theorem 4.2 bears some similarity to the argument showing decidability of containment of datalog programs in UCQs over arbitrary structures [Chaudhuri and Vardi 1992]. We show that satisfiability of $\mathcal{P} \wedge \neg \mathcal{Q}$ can be tested over special trees, so called canonical models, and the set of suitable encodings of canonical models satisfying $\mathcal{P} \wedge \neg \mathcal{Q}$ is recognized by a tree automaton.

So far our decidability results relied on bounding the number of labels in the models of $\mathcal{P} \wedge \neg \mathcal{Q}$; this requires a bound on the number of children. Over unranked trees we do the opposite: we use the fact that witnessing patterns of downward programs admit injective (up to ϵ-edges) homomorphisms into unranked trees, and we make the labels as different from each other as permitted by $\mathcal{P}$.

Definition 4.3. A tree is a **canonical model** for a satisfiable pattern π from TPQ($\downarrow, \downarrow^+, \epsilon$), if it can be obtained in the course of the following procedure:

(1) rename variables so that maximal ϵ-free subpatterns use disjoint sets of variables (do not change equalities within maximal ϵ-free subpatterns);
(2) unify labels of nodes connected with ϵ-edges (since π is satisfiable, whenever endpoints of an ϵ-edge have different labels, at least one is labeled with a variable: replace all occurrences of this variable with the other label);
(3) merge all nodes connected with ϵ-edges;
(4) substitute each variable with a fresh label;
(5) replace each $\downarrow^+$ edge with a sequence of nodes labeled with fresh labels, connected with $\downarrow$ edges.

An example of the construction described in Definition 4.3 is given in Fig. 4.

Note that a pattern from $\mathsf{TPQ}(\downarrow, \downarrow^+, \epsilon)$ represents a family of canonical models: fresh labels and, more importantly, the lengths of $\downarrow^+$ paths are chosen arbitrarily.

LEMMA 4.4. *Let $\mathcal{P}, Q \in$ D-Datalog$(\downarrow, \downarrow^+)$. Then $\mathcal{P} \wedge \neg\mathcal{Q}$ is satisfiable over unranked trees if and only if it is satisfiable in a canonical model of a witnessing pattern of $\mathcal{P}$.*

PROOF. Let t be a model for $\mathcal{P} \wedge \neg\mathcal{Q}$ and let π be the corresponding witnessing pattern for $\mathcal{P}$. Let π' be the pattern obtained from π by merging and unifying nodes connected with ϵ. By construction, π and π' are equivalent and their classes of canonical models coincide. It suffices to provide a canonical model of π' that does not satisfy $\mathcal{Q}$.

Let $t.v$ be the subtree of t rooted at node v. Consider a copy of t with $t.v$ duplicated, i.e., with an additional copy of $t.v$ also rooted at v. Then, the modified tree satisfies $\mathcal{Q}$ if and only if $t \models \mathcal{Q}$. The 'if' implication is obvious. For the 'only if' direction suppose that there is a witnessing pattern with a homomorphism h to the modified tree. Whenever h maps a pattern node to a node w in the additional subtree, we modify h to map that pattern node to the original copy of w in $t.v$. Since there is no sibling relation in the signature, the modified h preserves binary relations and remains a homomorphism.

Consequently, we can duplicate subtrees of t so that π' is matched injectively but $\mathcal{Q}$ remains not satisfied. The next step is to remove from t all nodes that are not in the image of π' and have no descendant in the image of π' (this is important because of $\downarrow^+$-edges in π'). The obtained tree t' is still a model for $\mathcal{P} \wedge \neg\mathcal{Q}$.

Up to relabeling, t' is also a canonical model for π'. Let $t_{\pi'}$ be a canonical model for π' where the lengths of paths between nodes connected with $\downarrow^+$-edges are the same as the lengths of the corresponding paths in t'. Trees $t_{\pi'}$ and t' are isomorphic up to the labeling, and the isomorphism is compatible with the injective homomorphism from π' to t'. Consequently, whenever two nodes in $t_{\pi'}$ have equal labels, by definition the corresponding nodes in π' have equal labels, and so their images in t' also have equal labels. Similarly, whenever a node in $t_{\pi'}$ has a label explicitly used in $\mathcal{P}$, the corresponding node in t' has the same label. This proves that $t_{\pi'} \models \mathcal{P} \wedge \neg\mathcal{Q}$. $\square$

Let N be the maximal number of variables in a rule in $\mathcal{P}$. Let $\mathsf{TPQ}^{\mathcal{P}}(\downarrow, \downarrow^+, \epsilon)$ be the class of patterns from $\mathsf{TPQ}(\downarrow, \downarrow^+, \epsilon)$ with branching bounded by N and labels coming from $\Sigma_{\mathcal{P}} \subseteq \Sigma$ (labels used explicitly in $\mathcal{P}$) or a fixed set $\Delta_0 \subseteq \Delta$ of size N. As $\mathcal{P}$ is downward, its witnessing patterns are elements of $\mathsf{TPQ}^{\mathcal{P}}(\downarrow, \downarrow^+, \epsilon)$.

Let $\Sigma_0 = \Sigma_{\mathcal{P}} \cup \Delta_0$. Patterns in $\mathsf{TPQ}^{\mathcal{P}}(\downarrow, \downarrow^+, \epsilon)$ can be viewed as trees over the alphabet $\Sigma_0 \times \{\downarrow, \downarrow^+, \epsilon\}$, where the first component is the label of the node in the pattern, and the second component determines the kind of edge between the node and its parent (in the root the second component plays no role, we may assume that is ϵ). Conversely, each tree over this alphabet corresponds to a pattern from $\mathsf{TPQ}^{\mathcal{P}}(\downarrow, \downarrow^+, \epsilon)$ (up to the second component of the root label). The technical core of Theorem 4.2 is the following lemma.

LEMMA 4.5. *The set of patterns $\pi \in \mathsf{TPQ}^{\mathcal{P}}(\downarrow, \downarrow^+, \epsilon)$ such that $\mathcal{P} \wedge \neg\mathcal{Q}$ is satisfied in a canonical model of π is recognized by a double exponential tree automaton, whose states and transitions can be enumerated in exponential working memory.*

PROOF. We first show how to recognize (encodings of) canonical models of patterns from $\mathsf{TPQ}^{\mathcal{P}}(\downarrow, \downarrow^+, \epsilon)$ that satisfy $\mathcal{P} \wedge \neg\mathcal{Q}$, and then explain how this gives the claim.

Encoding. Our goal is to have patterns over the alphabet $\Sigma_0 \times \{\downarrow, \epsilon\}$, because each pattern without $\downarrow^+$ corresponds to a unique canonical model (up to the choice of fresh labels). We take care of this by adding a new label $\$$ to the alphabet Σ_0. Suppose we want to have $v \downarrow^+ w$ in our pattern. We encode it by adding a path from v to w with nodes labeled by $(\$, \downarrow)$. From the point of view of canonical models, the intended meaning is that nodes labeled with $(\$, \downarrow)$ encode nodes with unique labels. We disallow $(\$, \epsilon)$,

because contracting ϵ-edges might violate uniqueness of represented labels. From now on, patterns are over $\Sigma_0 \times \{\downarrow, \epsilon\}$ with a special label $\$ \in \Sigma_0$.

Partial matchings. In order to evaluate programs on the encodings, our automaton will need to store information about partial homomorphisms from patterns to the encoded tree. For pattern $\kappa \in \mathsf{TPQ}(\downarrow, \downarrow^+)$, let $Pos(\kappa)$ be the set of all nodes of κ and all $\downarrow^+$-edges of κ. The building blocks of the states are **partial matchings** of κ, i.e., elements of

$$Part(\kappa) = (\Sigma_0 \cup \{\flat, \bot\})^{Pos(\kappa)}.$$

A partial matching $f \in Part(\kappa)$ will summarize information about a partial homomorphism from κ into the subtree s of the canonical model read so far by the automaton: $f(p) = \bot$ means that the homomorphism is not defined on p. If $f(p) \neq \bot$ for all p, we call f a **total matching**; it represents a total homomorphism that maps the root of κ to the root of s. More generally, if f maps the parent of some $p \in Pos(\kappa)$ to $\bot$ and $f(p) \neq \bot$, then p is mapped to the root of s by the homomorphism represented by f. We shall be only interested in partial homomorphisms whose domains are unions of subtrees of κ; that is, we assume that if $f(p) \neq \bot$ and q is a descendant of p in κ, then $f(q) \neq \bot$. This convention applies also to $\downarrow^+$-edges. A $\downarrow^+$-edge from u to v is treated as a virtual node whose parent is u and the unique child is v. If this virtual node is matched at the root of tree s, it means that the subtree rooted at v has been matched, but v is not mapped to the root of s, but somewhere below.

If $f(p) \in \Sigma_0$ then $f(p)$ is the label of the node to which p was mapped. If p is a $\downarrow^+$-edge the label does not matter, it only matters whether $f(p) \neq \bot$; we shall assume that $\downarrow^+$-edges are always mapped to $\bot$ or $\flat$. Let us now explain how we use the value $\flat$ for node positions. The automaton will sometimes go through an ϵ-edge, which is indicated by ϵ in the letter in the root of s. As explained before, even though some of the labels we remember in partial matchings may repeat above the ϵ-edge, in reality they represent different labels in the model: the only labels that are really the same are the labels from $\Sigma_{\mathcal{P}}$ and the label used in the root of s. Therefore, before we finish processing the node s, we shall replace all other labels in f with $\flat$, which means that these positions of κ were mapped to nodes with labels that will never appear again in the model. Suppose f maps a node v from κ to $\bot$ but κ requires this node to have the same label as a different node already mapped to $\flat$. Then we can forget f because we will never find the label for v outside of s.

Automaton evaluating a downward program $\mathcal{R}$ on encodings. The states of automaton $\mathcal{A}_{\mathcal{R}}$ are subsets of

$$Part(\mathcal{R}) = \bigcup_{r \in \mathcal{R}} Part(\pi_r)$$

plus an accepting state $\top$ (the state space is doubly exponential). The automaton essentially computes partial matchings of patterns corresponding to the rules of $\mathcal{R}$ in the currently read subtree, and uses them to find a matching of a witnessing pattern.

Suppose that the currently processed node v has label (a, ϵ). In this case we shall propagate up the tree partial matchings at the children of v, not at v itself. Let M be the union of states in v's children (possibly none), closed under compatible unions of partial matchings. Recall that positions mapped to $\flat$ in different partial matchings of pattern π_r in fact get different labels; this gives incompatibility if they are labeled with the same variable in π_r, and such partial matchings cannot be combined. The state in v is M, except that each $b \in \Sigma_0 \setminus (\Sigma_{\mathcal{P}} \cup \{a\})$ used in a matching should be replaced with $\flat$, because any future occurrences of b will represent different labels.

If the label of the currently processed node v is $(a, \downarrow)$, we also start from the union M of states in v's children, closed under compatible unions of partial matchings. This

gives partial matchings at the children of v. Now, we would like to compute partial matchings at v. Essentially, we should extend each matching $f \in M$ of pattern π_r to v, whenever possible. How do we do this?

Assume position p is an unmatched node in π_r. We can extend f to p if and only if

— each descendant of p is matched; and
— either p has label a, or p has label $x \in \Delta$ and each *matched* node position p' with the same label x is mapped to a.

If we choose to extend, we set $f(p) = a$, unless $a = \$$; since $\$$ represents a label that occurs nowhere else, if $a = \$$ we set $f(p) = \flat$. If at least one of p's children is a matched node position, we must extend f to p; if we cannot, we discard f.

If position p is an unmatched $\downarrow^+$-edge, we can (and must) extend f to p, if and only if the unique child of p is matched. As we have explained, we map p to $\flat$. Note also that if a $\downarrow^+$-edge p is matched (at some child of v) and its parent position is not matched, we can also match p at v: the matching f is not modified at p, but it represents a different homomorphism.

Let M' be the set of all possible extensions of matchings from M. Clearly, M' correctly represents all partial homomorphisms from patterns π_r to the read subtree. But we are interested in homomorphisms of witnessing patterns, not just of patterns π_r. We cannot remember the whole matchings of witnessing patterns, because they can be arbitrarily large. The trick is to forget parts of it as soon as they are no longer needed. Consider a node u in π_r that corresponds to a variable occurring in some intensional atoms of rule r. It is clear, that if node u is to be matched at the currently processed node v of the tree, each of these intensional atoms should be witnessed by a total matching (at v) of a pattern corresponding to one of its rules. Note that this is a recursive condition, as these patterns may need other patterns matched at v. Thus, the state after processing v should contain exactly those partial matchings that satisfy these requests for each node to be matched at v. We can compute them from $\mathcal{R}$ and M' in PTIME (simple reachability), which means time exponential in the size of $\mathcal{R}$.

Automaton $\mathcal{A}_{\mathcal{R}}$ accepts if it finds a total matching of the pattern corresponding to the goal rule of $\mathcal{R}$.

Recognizing patterns whose canonical models satisfy $\mathcal{P} \wedge \neg\mathcal{Q}$. We define the standard product automaton $\mathcal{A}_{\mathcal{P}} \times \overline{\mathcal{A}_{\mathcal{Q}}}$, where $\overline{\mathcal{A}_{\mathcal{Q}}}$ denotes the complement of $\mathcal{A}_{\mathcal{Q}}$ (it is deterministic, so there is no blow-up). This automaton recognizes whether $\mathcal{P} \wedge \neg\mathcal{Q}$ is satisfied in the encoded model. We modify this automaton to work on the original encodings of the models, i.e., on patterns from $\mathsf{TPQ}^{\mathcal{P}}(\downarrow, \downarrow^+, \epsilon)$: upon reading label $(a, \downarrow^+)$, the automaton first updates the states as if the label was $(a, \downarrow)$, and then performs ϵ-transitions simulating arbitrary long paths of nodes labeled with $(\$, \downarrow)$. This introduces non-determinism, but it does not matter since we already have an automaton for the whole query $\mathcal{P} \wedge \neg\mathcal{Q}$. □

To test satisfiability of $\mathcal{P} \wedge \neg\mathcal{Q}$, we build the automaton given by Lemma 4.5 and test its emptiness. Since emptiness can be tested in PTIME, this is a 2-EXPTIME algorithm.

If $\mathcal{P}$ is linear ($\mathcal{Q}$ may be non-linear), we can do better. As proof trees of linear programs are words, their witnessing patterns have a very particular shape: one main branch (corresponding to the recursive calls of intensional predicates) and small subtrees off this branch with at most N branches (fragments of patterns corresponding $\mathcal{P}$'s rules). Over such trees emptiness of a given automaton $\mathcal{B}$ can be tested by a nondeterministic algorithm in space $\mathcal{O}(N \log |\mathcal{B}|)$. Indeed, the claim is well known (and easy to prove) for trees with at most N branches (see e.g., [Neven and Schwentick 2006]). In our case, the algorithm can keep guessing the main path bottom-up, to-

gether with the states in the nodes just off the path, and use the previous result as a subprocedure to test if $\mathcal{B}$ can accept from these states.

Generating the automaton from Lemma 4.5 on the fly in exponential working memory, we can test its emptiness on trees of special shape in EXPSPACE. A natural generalization of the claim above gives EXPSPACE algorithm for *nested linear programs*, where any predicate defined by a linear subprogram can be used freely outside of this subprogram. Nested linear programs are a more robust class then linear programs, e.g., they are closed under conjunction.

4.2. Linear child-only programs

For child-only programs, we get decidability if we assume linearity.

THEOREM 4.6. *Over unranked trees containment for* L-Datalog($\downarrow$) *is in 3-*EXPTIME*; containment of* L-Datalog($\downarrow$) *programs in* UCQ($\downarrow$) *queries is 2-*EXPTIME*-complete.*

The lower bound is proved in Section 6 (Proposition 6.4); the reduction is the same as for L-Datalog($\downarrow$) over ranked trees (Theorem 3.3). The general strategy to prove the upper bound is as for downward programs, but this time we face a new difficulty. The patterns corresponding to rules of L-Datalog($\downarrow$) programs need not be tree patterns any more, but without loss of generality we can assume that they are: since each rule is connected, by merging nodes sharing a child one obtains a pattern from TPQ($\downarrow,\uparrow$). Witnessing patterns are then elements of TPQ($\downarrow,\uparrow,\epsilon$): sequences of TPQ($\downarrow,\uparrow$) patterns connected with ϵ-edges between *out* nodes and *in* nodes. The problem is that for such patterns, before demanding injective homomorphisms, we need to merge not only nodes connected with ϵ-edges, but also (again) nodes sharing a child.

The first step to fix this is to adjust canonical models. One way to do this is to modify accordingly item 3 in Definition 4.3 (item 5 becomes void), but we prefer to give a different definition (yielding the same object), that reflects more accurately the way automata look at canonical models.

Definition 4.7. Fix a pattern $\pi \in \mathsf{TPQ}(\downarrow,\uparrow,\epsilon)$ and rename variables so that maximal ϵ-free subpatterns of π use disjoint sets of variables. The **canonical model** t_π and the witnessing **canonical homomorphism** $h_\pi\colon \pi \to t_\pi$ are constructed by an iterative procedure processing the nodes of π in a top down fashion. With each processed node, the partially constructed t_π and h_π are extended, and some variables of π may get unified or substituted with constants from Σ (formally, an equivalence relation θ_π is maintained over the labels of π—variables and constants). We begin with a tree consisting of a single node v_0 and a partial homomorphism $h_\pi(root(\pi)) = v_0$. While there is an unprocessed node x, whose parent y is processed and $h_\pi(y) = v$, we perform one of the following, depending on the label of edge (x, y):

($\downarrow$) add a new node u with label $lab_\pi(x)$ as the *last* child of v, and set $h_\pi(x) = u$;
(ϵ) unify $lab_{t_\pi}(v)$ and $lab_\pi(x)$ in π and in t_π, and set $h_\pi(x) = v$;
($\uparrow$) if v has a parent u, unify $lab_{t_\pi}(u)$ and $lab_\pi(x)$, and set $h_\pi(x) = u$,
if v has no parent, add parent u with label $lab_\pi(x)$, and set $h_\pi(x) = u$.

If at any point unification of two constants is attempted, the procedure fails. Otherwise, before returning the result, we replace each variable in t_π with a fresh constant.

Sibling order in case ($\downarrow$) is specified intentionally, for the purpose of the automata construction in Lemma 4.11 below. An example of the construction is given in Fig. 5.

LEMMA 4.8. *Let π, ρ be patterns from* TPQ($\downarrow,\uparrow,\epsilon$)*, and let π be satisfiable.*

(*1*) *There exists a canonical model t_π, unique up to the choice of fresh labels and the sibling order, and a witnessing homomorphism $h_\pi\colon \pi \to t_\pi$.*

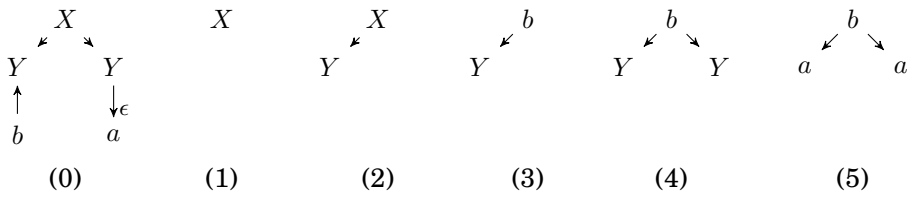


Fig. 5: Consecutive steps in transforming a pattern (0) into a canonical model (5). Nodes are processed in the prefix order.

(2) *If t_π satisfies ρ, then each tree satisfying π also satisfies ρ (unless ρ uses one of the fresh constants from the construction of t_π).*

PROOF. Each unification of labels in the construction described in Definition 4.7 is induced by unifying labels of nodes that must be mapped to the same tree node (by the semantics of patterns). Consequently, if pattern π can be mapped to some tree, unification of two different constants will never be attempted during the construction. Hence, for a satisfiable π the procedure always successfully builds t_π and $h_\pi \colon \pi \to t_\pi$. The order in which the nodes of π are processed only influences the sibling order in t_π.

Before the final substitution of variables with fresh constants, t_π can be seen as a pattern $\pi' \in \mathsf{TPQ}(\downarrow)$. By construction, π' is equivalent to π. As long as ρ does not use the fresh constants replacing variables of π' in t_π, each homomorphism $g \colon \rho \to t_\pi$ induces a homomorphism $\hat{g} \colon \rho \to \pi'$. Now, if a tree t satisfies π, it also satisfies π' with some witnessing homomorphism $g' \colon \pi' \to t$, which gives a homomorphism $\hat{g} \circ g' \colon \rho \to t$. □

As an immediate corollary from Lemma 4.8 we obtain an analogue of Lemma 4.4.

COROLLARY 4.9. *Let $\mathcal{P}, \mathcal{Q} \in \mathsf{Datalog}(\downarrow)$. $\mathcal{P} \wedge \neg\mathcal{Q}$ is satisfiable over unranked trees iff it is satisfiable in a canonical model of a witnessing pattern of $\mathcal{P}$.*

Thus, like before, deciding satisfiability of $\mathcal{P} \wedge \neg\mathcal{Q}$ amounts to testing existence of a witnessing pattern π for $\mathcal{P}$ whose canonical model does not satisfy $\mathcal{Q}$. This is also more involved than for downward programs: nodes arbitrarily far apart in π may represent the same node of the canonical model. An automaton cannot compute this correspondence; we need to make it explicit. In place of direct encodings of patterns, we shall use encodings that combine information about π, t_π, and h_π in a single tree. The idea is to encode π and h_π as an additional labeling of t_π. But, since unboundedly many nodes of π can be mapped to the same node of t_π, there is no natural way to do it with a finite alphabet. The solution is to restrict the representation to the main branch of π, which determines the way $\mathcal{P}$ traverses the tree.

Being an L-$\mathsf{Datalog}(\downarrow)$ program, $\mathcal{P}$ traverses the tree by moving from node to node checking some local conditions, until it reaches an accepting state: a satisfied non-recursive rule. Each time, the node to move to is determined by what we call the spine of the current rule r, which can be defined in terms of the corresponding pattern π_r as follows. The **spine** of $\pi \in \mathsf{TPQ}(\downarrow, \uparrow)$, is formed by the shortest path connecting $\mathit{in}(\pi)$ to $\mathit{out}(\pi)$. The remaining nodes form the **limbs** of π; they are a collection of subpatterns, each attached to a spine node. If the *in* and *out* nodes in π_r coincide—which by convention includes all non-recursive rules—we call r **stationary**. For stationary rules the spine is trivial: it consists of the *in* node and no edges.

The partition into spine and limbs can be lifted to witnessing patterns. Since program $\mathcal{P}$ is linear, all the ϵ-edges in a witnessing pattern π are on a single branch: this branch, from the root down to the last ϵ-edge, is the spine of π. Note that the spine

of π is the concatenation of the spines of the maximal ϵ-free subpatterns of π, with ϵ-edges inserted between them (the spine of the last one consists of a single node, as the corresponding rule of $\mathcal{P}$ is non-recursive).

Let ρ be the spine of π. Observe that canonical model t_ρ is isomorphic to the subtree $h_\pi(\rho)$ of t_π, and that h_ρ coincides with h_π restricted to ρ (up to this isomorphism). We shall focus on ρ, t_ρ, h_ρ. Note that there is still no bound on the number of nodes of ρ mapped to a single node of t_ρ. But h_ρ induces a mapping from edges of ρ to edges of t_ρ, and the number of non-ϵ edges mapped to a single edge of t_ρ is bounded by 2. Indeed, in the procedure building t_ρ for any pattern ρ that does not branch, each edge is traversed once when it is created, upwards or downwards. If it was created while going down, it may be traversed once more, going up. Once we go up along an edge, we never come back, because whenever we go down, we move to a freshly created node.

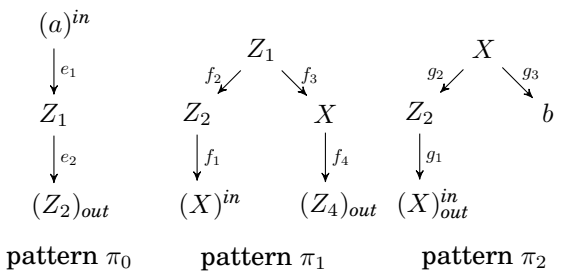
Consequently, we can encode ρ, t_ρ, h_ρ as a tree obtained from t_ρ by erasing node labels, and labeling its edges with the edges of ρ mapped to them by h_ρ. More precisely, the labels come from the alphabet

$$\big(\mathit{Spine}(\mathcal{P}) \cup \{\top\}\big) \times \big(\mathit{Spine}(\mathcal{P}) \cup \{\top\}\big)\,,$$

where $\mathit{Spine}(\mathcal{P})$ is the disjoint union of the sets of spine edges of π_r, for all rules $r \in \mathcal{P}$. The two coordinates describe traversing the edge downwards and upwards, respectively; $\top$ is used for edges traversed only once.

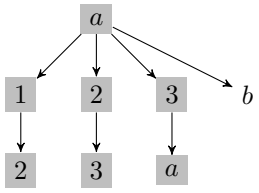
Example 4.10. Consider the program on the left and the patterns on the right cor-

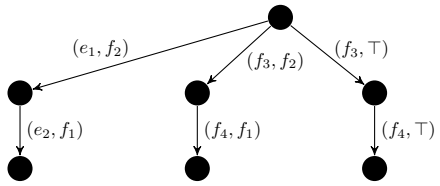
$$\begin{aligned}
\underline{P(X)} &\leftarrow X{\downarrow}Z_1, Z_1{\downarrow}Z_2,\\
&\qquad a(X), Q(Z_2)\\
Q(X) &\leftarrow Z_1{\downarrow}Z_2, Z_2{\downarrow}X, Z_1{\downarrow}Z_3, Z_3{\downarrow}Z_4,\\
&\qquad X \sim Z_3, Q(Z_4)\\
Q(X) &\leftarrow Z_1{\downarrow}Z_2, Z_2{\downarrow}X, Z_1{\downarrow}Z_3,\\
&\qquad X \sim Z_1, b(Z_3)
\end{aligned}$$



pattern π_0 pattern π_1 pattern π_2

responding to its rules. The canonical model t_π for witnessing pattern $\pi = \pi_0\pi_1\pi_1\pi_2$ is shown below, on the left; the highlighted nodes belong to t_ρ, where ρ is the spine of π.





To the right is the encoding of ρ; note how the use of $\top$ reflects the fact that the spine of π_2 contains no edges.

Theorem 4.6 is an immediate corollary from the following lemma. Formally, one can turn edge labelled trees to node labelled trees by moving the label from the edge to the lower of the two nodes it connects; the root gets label $(\top, \top)$.

LEMMA 4.11. *The set of encodings of spines of witnessing patterns of $\mathcal{P}$, whose canonical models satisfy $\mathcal{P} \wedge \neg\mathcal{Q}$, is regular. The recognizing (unranked) tree automaton can be computed in* 3-EXPTIME *in general, and in* 2-EXPTIME *if* $Q \in \mathsf{UCQ}(\downarrow)$.

PROOF. The encoding we chose has one drawback: it is not injective. While the missing information about the limbs and the corresponding nodes of the canonical model can be easily recovered (as long as we know the program $\mathcal{P}$), keeping only $\uparrow$ and $\downarrow$ edges means that the information about stationary rules is lost completely.

The automata construction is easier to describe for an extended encoding, where the stationary rules are listed explicitly: between any two siblings of the original encoding we allow an arbitrary sequence of leaves with edges leading to them labelled with the *in*-nodes of stationary rules. Obviously, an automaton recognizing these extended encodings can be turned into an automaton recognizing the original encodings: the new automaton simply guesses the stationary rules as it goes.

The automaton has to check that

(1) the input tree is an extended encoding of the spine of a witnessing pattern π for $\mathcal{P}$;
(2) the canonical model t_π exists (equivalently, pattern π is satisfiable) ; and
(3) $\mathcal{Q}$ is not satisfied in the canonical model t_π.

Unlike in Lemma 4.5, where we built independent automata for $\mathcal{P}$ and $\mathcal{Q}$ and took a product automaton for $\mathcal{P} \wedge \neg\mathcal{Q}$, here we construct the whole automaton at once. However, the components are to some extent independent and we describe them separately.

To verify (1), it suffices to check two sub-conditions. First, that pattern edges in labels always go down in the first component and up in the second component. Second, that reading the labels in the whole tree in the depth-first order starting from the root—the first component when going down and the second when going up (for stationary rules the second component is empty)—we obtain a sequence that can be turned into the spine of a witnessing pattern by erasing the initial and final $\top$ symbols and inserting ϵ-edges between consecutive rules. Both sub-conditions can be easily checked with an automaton of size polynomial in the size of $\mathcal{P}$. From now on we assume that the automaton reads an extended encoding of the spine ρ of a witnessing pattern π (in which maximal ϵ-free subpatterns use disjoint sets of variables). Verifying the remaining two conditions is much more involved.

Checking condition (2) amounts to verifying that the construction of the canonical model described in Definition 4.7 succeeds, that is, unification of two constants from $\Sigma_\mathcal{P}$ (the set of labels used explicitly in $\mathcal{P}$) is never attempted. Towards this goal, observe a certain compositionality of canonical models. Consider two patterns π_0, π_1, possibly sharing variables. Slightly abusing notation, we shall write $\pi_0\pi_1$ for the pattern obtained by identifying the *out* node of π_0 and the *in* node of π_1 (without relabeling the shared variables). How is $t_{\pi_0\pi_1}$ related to t_{π_0} and t_{π_1}? The canonical models t_{π_0} and t_{π_1} come together with equivalence relations θ_{π_0} and θ_{π_1} describing the unification of labels in π_0 and π_1. Since π_0 and π_1 may share variables, θ_{π_0} and θ_{π_1} do not have disjoint domains and after taking their union we need to close it under transitivity to obtain the correct equivalence relation over the labels of $\pi_0\pi_1$. This may unify some more labels and possibly result in a failure if unification of two constants is attempted. If this succeeds, it follows easily from Definition 4.7 that $t_{\pi_0\pi_1}$ can be obtained by merging t_{π_0} and t_{π_1} along the paths going upwards from the nodes $h_{\pi_0}(\mathit{out}(\pi_0))$ and $h_{\pi_1}(\mathit{in}(\pi_1))$. Again, this will typically induce some unifications of labels on the two paths, which may result in a failure. Thus, we have an operation on canonical models that corresponds to concatenation of patterns. Using this operation we can build canonical models for a pattern by composing canonical models for its fragments (by the uniqueness of the canonical model, the result does not depend on the way we partition the pattern).

To build (an abstraction of) the canonical model t_π the automaton will process the encoding of ρ bottom-up and compose (abstractions of) the canonical models for fragments of π induced by the infixes of ρ. The subtree of the encoding of ρ rooted at a node u corresponds to the infix ρ_u of ρ, determined by the first and last component of the label of the edge leading to node u from its parent. Hence, when the automaton is in node v of the encoding, and has already processed subtrees rooted in the first k of the v's children $v_1, v_2, \ldots, v_m$, then it has seen an infix $\rho' = \rho_{v_1}\rho_{v_2}\ldots\rho_{v_k}$ of the spine ρ. For any infix σ of ρ, let $\pi[\sigma]$ denote the fragment of π consisting of σ and all the limbs of π attached to σ (to avoid duplicates, the limbs attached to the last node of σ are included only if it is also the last node of the spine of some π_r). The automaton has to retain sufficient information about the canonical model $t_{\pi[\rho']}$ to test if it can be further composed with the canonical model for $\pi[\rho_{v_{k+1}}]$ to obtain the canonical model for $\pi[\rho_{v_1}\rho_{v_2}\ldots\rho_{v_k}\rho_{v_{k+1}}]$.

From the way we compose canonical models it follows immediately that it suffices to store the equivalence classes of labels shared by the canonical models and those represented in the paths along which we merge the canonical models. Given that subpatterns corresponding to rules use disjoint sets of variables, it suffices to store the equivalence classes of labels in the first and last rule of ρ'. Since ρ' was read from a sequence of subtrees rooted at subsequent siblings in the encoding, the first and last node of ρ' are mapped to the same node u of the canonical model $t_{\pi[\rho']}$, and all others are mapped to the subtree rooted at u. Consequently, the limbs of $\pi[\rho']$ may reach at most n levels up from u, where n is the maximal size of a rule in program $\mathcal{P}$. This means that the path from u up to the root has length at most n. But this path is precisely the one along which $t_{\pi[\rho']}$ will be merged with the canonical model for $\pi[\rho_{v_{k+1}}]$; we shall refer to it as the *merging path*. Summing up, we need to represent at most $3n$ equivalence classes of labels: n for the merging path, n for the first rule of ρ', and n for the last rule of ρ'. The classes containing a constant from $\Sigma_\mathcal{P}$ will be represented by this value (there is at most one such constant in each class), the remaining classes will be represented with fresh labels from a fixed set $\Delta_0 \subseteq \Delta$, say $\Delta_0 = \{1, 2, \ldots, 3n\}$. The information about the first and last rule is stored in the form of partial matchings. A *partial matching* of pattern π_r (corresponding to a rule r) is a function

$$f\colon vertices_{\pi_r} \to \Sigma_\mathcal{P} \cup \Delta_0 \cup \{\flat, \bot\}$$

where $\bot$ indicates that a node is not processed yet, and $\flat$ means it has been processed, but the equivalence class of its label will not appear in the tree any more; we write $Part(\pi_r)$ for the set of all partial matchings of π_r. Altogether, the component of the state-space used by the satisfiability test is

$$Part(\mathcal{P})^2 \times (\Sigma_\mathcal{P} \cup \Delta_0)^n$$

where $Part(\mathcal{P}) = \bigcup_{r\in\mathcal{P}} Part(\pi_r)$ and $(\Sigma_\mathcal{P} \cup \Delta_0)^n$ represents the equivalence classes of the labels on the merging path.

Let us see that this information can be updated as the automaton processes the encoding of the spine ρ. It is convenient to diverge from the formal definition given in the preliminaries and let the automaton evaluate subtrees and subforests associated with subsequent siblings. More precisely, suppose a node v has children $v_1, v_2, \ldots, v_m$. The state for a subforest associated to $v_1, v_2, \ldots, v_{k+1}$ is obtained based on the state for the subforest associated to $v_1, v_2, \ldots, v_k$ and the state for the subtree rooted at v_{k+1}. The state for the subtree rooted at v is obtained based on the state for the subforest associated to $v_1, v_2, \ldots, v_m$ and the label on the edge from v's parent to v. Assuming this convention, combining the information computed for $\rho' = \rho_{v_1}\rho_{v_2}\ldots\rho_{v_k}$ and $\rho_{v_{k+1}}$ into the information for $\rho_{v_1}\rho_{v_2}\ldots\rho_{v_k}\rho_{v_{k+1}}$ amounts to performing the unifications induced by variables shared by the last pattern of ρ' and the first pattern of $\rho_{v_{k+1}}$, and the

unifications induced by identifying the corresponding nodes of the merging paths, and then keeping the information about the (unified) merging path, the first rule of ρ' and the last rule of $\rho_{v_{k+1}}$. When the automaton is done with all the children of v, it has to turn the information computed for $\rho_{v_1}\rho_{v_2}\dots\rho_{v_m}$ into the information for $\rho_v = e\rho_{v_1}\rho_{v_2}\dots\rho_{v_m}e'$, where (e, e') is the label on the edge from v's parent to v. This is done by combining the information about the canonical models for $\pi[e]$, $\pi[\rho_m]$, and $\pi[e']$, just like above; if v happens to be a leaf, we take the information about the canonical model for $\pi[ee']$. As the final step before moving up we shift the stored path: we cut off the lowest node and add a new one on top with a label from Δ_0 that is not used anywhere else (the size of Δ_0 guarantees there is one). If v is a leaf representing a stationary rule r, we return the information about the canonical model for π_r (without shifting up). Note that the whole computation involves the information stored and one or two fragments of patterns corresponding to rules of $\mathcal{P}$, so it can be precomputed and encoded in the transition relation of the automaton.

Condition (3) requires checking that $\mathcal{Q}$ is not satisfied in the canonical model t_π. To achieve this, automaton $\mathcal{A}_{\mathcal{P}\wedge\neg\mathcal{Q}}$ will compute all possible subwords of proof words (proof subwords, for short) for $\mathcal{Q}$ satisfied in the part of t_π analyzed so far. More precisely, being in node v and having processed first k of its m children $v_1, v_2, \dots, v_m$, the automaton will have computed proof subwords of $\mathcal{Q}$ that can be satisfied in the canonical model $t_{\pi[\rho']}$, where $\rho' = \rho_{v_1}\rho_{v_2}\dots\rho_{v_k}$.

To represent information about patterns corresponding to the rules forming proof subwords, we use partial matchings like in the downward case, but with an important difference. Before, we were always matching the patterns in the currently processed node v. Now, the node v does not correspond to the root of $t_{\pi[\rho']}$, but to some node u which can be as low as n levels deep in $t_{\pi[\rho']}$. Moreover, the part of the tree above u will still evolve, as more limbs of π reaching up are incorporated. Consequently, we cannot assume that patterns are matched at the root of $t_{\pi[\rho']}$ either. We need to allow matching at any node on the path in $t_{\pi[\rho']}$ from u upwards (the merging path). To represent this, we use numbers between 0 and n determining how many nodes above u a node of the pattern is matched, and $\flat$ for nodes mapped off the merging path starting in u. Let $Match(\pi_r)$ be the set of functions

$$f\colon vertices_{\pi_r} \to \big(\Sigma_{\mathcal{P}} \cup \Delta_0 \cup \{\flat\}\big) \times \{0, 1, \dots, n, \flat\} \cup \{\bot\}$$

and define

$$Match(\mathcal{Q}) = \bigcup_{r\in\mathcal{Q}} Match(\pi_r)\,.$$

As proof subwords for $\mathcal{Q}$ can be arbitrarily long, automaton $\mathcal{A}_{\mathcal{P}\wedge\neg\mathcal{Q}}$ cannot keep them directly in memory. The first idea is to represent them with pairs of partial matchings, corresponding to the first and the last rule of the proof subword. When processing the tree, the automaton would extend these partial matchings locally, and compose compatible pairs of partial matchings, dropping the complete middle one. Unfortunately, this is not enough. Automaton $\mathcal{A}_{\mathcal{P}\wedge\neg\mathcal{Q}}$ collects information about t_π by processing the spine ρ of π. While existence of each node of t_π is triggered by a nearby spine node, equalities of labels in t_π can be enforced by very distant spine nodes. Indeed, incorporating a limb can unify some labels on the merging path, and these labels could have been used anywhere in t_π. This of course affects the proof subwords satisfied in t_π. Consequently, even intermediate patterns of a proof subword that seemingly refer only to distant parts of t_π may not be fully matched: they may require label equality (or presence of a constant), that will be enforced only after incorporating into t_π some limbs attached to the yet unprocessed part of ρ. The automaton $\mathcal{A}_{\mathcal{P}\wedge\neg\mathcal{Q}}$ has to assume that all these missing requirements will be met later, when additional information

about t_π is provided. Therefore, for every proof subword of $\mathcal{Q}$, we store not only partial matchings of the first and last pattern, but partial matchings of all the patterns in the proof subword. As the automaton collects information about t_π, those partial matchings become less partial, and are dropped as soon as they become complete matchings; if any of them becomes impossible to complete, the whole proof subword is discarded. While the proof subwords have unbounded length, the corresponding sets of partial matchings are contained in the fixed finite set $Match(\mathcal{Q})$. This is possible because some patterns may be already matched completely somewhere lower in the tree, and some patterns may yield identical partial matchings (the proof subword could then be shortened, but we do not care). Thus, each proof subword is represented as a set of partial matchings of its rules, with the ones for the first and last rule distinguished. Since $\mathcal{A}_{\mathcal{P}\wedge\neg\mathcal{Q}}$ needs to represent all proof subwords satisfied in the processed part of t_π, the additional component of the state-space is

$$\mathbb{P}\big(Match(\mathcal{Q}) \times \mathbb{P}(Match(\mathcal{Q})) \times Match(\mathcal{Q})\big)$$

If $\mathcal{Q} \in \mathsf{UCQ}(\downarrow)$, each proof word consists of a single rule, so we use $\mathbb{P}(Match(\mathcal{Q}))$ instead.

Finally, compared to previous uses, we allow more flexibility in partial matchings. Nodes with constant labels can be also assigned variables (but not different constants), and nodes with the same variable label can be assigned different labels, as long as there is at most one constant among them. On the other hand, we shall still consider only partial matchings in which nodes assigned values different from $\bot$ form a connected subpattern. Moreover, if a partial matching of pattern π_r represents an intermediate rule of the proof subword, this subpattern contains the whole spine of π_r. A partial matching is considered complete, when each node with constant label is assigned this label, and all nodes with the same variable label are assigned the same label (not $\bot$).

Let us describe how this information is maintained. Using the notation introduced for (2), suppose we have computed the information about the canonical models $t_{\pi[\rho']}$ and $t_{\pi[\rho'']}$, where $\rho' = \rho_{v_1}\rho_{v_2}\dots\rho_{v_k}$ and $\rho'' = \rho_{v_{k+1}}$, and about the proof subwords of $\mathcal{Q}$ they satisfy. We have already seen how to compute the information about $t_{\pi[\rho'\rho'']}$, necessary to verify condition (2). To compute the information about proof subwords of $\mathcal{Q}$ satisfied in $t_{\pi[\rho'\rho'']}$, we perform the following three steps.

(1) Computing the information about $t_{\pi[\rho'\rho'']}$ involved performing some unifications of labels in $t_{\pi[\rho']}$ and $t_{\pi[\rho'']}$. The information about proof subwords of $\mathcal{Q}$ refers to the same labels, so we apply all these unifications there as well (this typically makes some partial matchings more complete).
(2) We take the union of the two sets representing proof subwords satisfied in $t_{\pi[\rho']}$ and $t_{\pi[\rho'']}$, and close it by concatenation as follows. We can concatenate (f, H, g) and (f', H', g') if g and f' are compatible partial matchings of the same pattern: they agree on the nodes assigned values different from $\bot$ and at least one of these nodes is mapped to the merging path (that is, the second component of the assigned value is not $\flat$). The result of the concatenation is
$$(f, H \cup H' \cup \{g \cup f'\}, g')\,,$$
where $(g \cup f')(x)$ equals $g(x)$ if $g(x) \neq \bot$, and $f'(x)$ otherwise.
(3) When computing information about $t_{\pi[\rho'\rho'']}$, some labels are forgotten: the ones represented only in the matching of the last pattern of ρ' or the first pattern of ρ''. These labels will not occur in the tree anymore (even if the values representing them in the automaton do). We discard all (f, H, g) such that H contains a partial matching that assigns a constant or a forgotten label to a node with a different constant label, or assigns either two different constants or a forgotten label and any

other value (including $\perp$) to nodes with the same label. In the remaining (f, H, g) we replace with $\flat$ all occurrences of forgotten labels.

Assume now the automaton has processed all the children of node v, obtaining information about the canonical model $t_{\pi[\rho']}$ for $\rho' = \rho_{v_1}\rho_{v_2}\dots\rho_{v_m}$, and about the proof subwords for $\mathcal{Q}$ satisfied in $t_{\pi[\rho']}$. In order to incorporate the information about the new edges e, e' (read from the label on the edge from v's parent to v) and the limbs connected to them, we repeat the steps (1)-(3) above twice: first for $t_{\pi[e]}$ and $t_{\pi[\rho']}$, and then for $t_{\pi[e\rho']}$ and $t_{\pi[e']}$; the information about the proof subwords of $\mathcal{Q}$ satisfied in $t_{\pi[e]}$ and $t_{\pi[e']}$ can be precomputed. If v is a leaf, we take the precomputed information about $t_{\pi[ee']}$. After this is done, it remains to shift the whole representation up by one level. This amounts to decreasing by one the numbers used in the partial matchings (0 is replaced with $\flat$). It may also involve forgetting the label of the lowest node of the stored path, so step (3) above has to be repeated. If v is a leaf representing a stationary rule r, we just take the precomputed information for t_{π_r} (without shifting up).

If at any point $\mathcal{A}_{\mathcal{P}\wedge\neg\mathcal{Q}}$ finds a complete proof word for $\mathcal{Q}$, it rejects.

The number of states of $\mathcal{A}_{\mathcal{P}\wedge\neg\mathcal{Q}}$ is triple exponential in $|\mathcal{Q}|$. Notice that the third exponential comes from the double power-set used for the evaluation of $\mathcal{Q}$. In the case of containment in $\mathsf{UCQ}(\downarrow)$, the size of the automaton drops to double exponential in $|\mathcal{P}|+|\mathcal{Q}|$. The construction can be easily implemented within declared time bounds. □

5. UNDECIDABILITY RESULTS

In this section we prove the three undecidability results claimed in Section 3 and Section 4: for $\mathsf{LD\text{-}Datalog}(\downarrow,\downarrow^+)$ over words and ranked trees, and for $\mathsf{Datalog}(\downarrow)$ over unranked trees.

We begin with the first item of Proposition 3.1, that is, $\mathsf{LD\text{-}Datalog}(\downarrow,\downarrow^+)$ over words. As each Boolean $\mathsf{UCQ}(\downarrow,\downarrow^+)$ query can be expressed as an $\mathsf{LD\text{-}Datalog}(\downarrow,\downarrow^+)$ program, Proposition 5.1 below is all we need to show. In the proof we use an RTPQ-like syntax for programs on words. In acronyms and notations we replace letter T with letter W (for word): we speak of WPQs, RWPQs, $\mathsf{RWPQ}(\rightarrow,\rightarrow^+)$ etc. For readability, in such queries we write the arrows horizontally, and we often skip the symbol $\rightarrow$ entirely. Let us underline that the RWPQs are always considered on models restricted to words.

PROPOSITION 5.1. *Over words, containment of* $\mathsf{LD\text{-}Datalog}(\downarrow,\downarrow^+)$ *programs in* $\mathsf{UCQ}(\downarrow,\downarrow^+)$ *queries is undecidable.*

PROOF. We show undecidability via a reduction from the tiling problem: Given

— a finite set of tiles K,
— a horizontal correctness relation $H \subseteq K \times K$,
— a vertical correctness relation $V \subseteq K \times K$,
— an initial tile τ_{ini},
— a final tile τ_{fin},

decide if there exist natural numbers M, N such that there is a tiling of the $M \times N$ grid that has τ_{ini} in cell $(1,1)$, τ_{fin} in cell (M,N), and all pairs of adjacent tiles satisfy the appropriate relation, H or V.

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

We shall construct queries $\varphi, \psi \in \mathsf{RWPQ}(\rightarrow,\rightarrow^+)$. Query φ will describe constraints enforced by tiling relations, initial configuration and ending tile. Query ψ will be a

union of conjunctive queries and $\neg\psi$ will be used to guarantee that each matched word is a proper encoding of some tiling.

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

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

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

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

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

The positive query φ is defined as

$$\varphi_{\textit{first-tile}} \cdot (\varphi_{\textit{row}})^{*} \cdot \varphi_{\textit{last-row}} \tag{1}$$

where

$$\varphi_{\textit{first-tile}} = (\#)^{in}_{out}\, 1\, 2\, 1\, \tau_{ini}$$

initiates the first tile,

$$\varphi_{\textit{last-row}} = \left[(\#)^{in}\, (R)_{out}\right] \cdot \beta^{*} \cdot \left[(R)^{in}\, R'\, C\, \tau_{\textit{fin}}\, (\#)_{out}\right]$$

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

$$\varphi_{\textit{row}} = \bigvee_{(\tau,\tau'')\in V} \bigvee_{\tau'\in K} \left[(\#)^{in}\, (R)_{out}\right] \cdot \alpha^{*} \cdot \left[(R)^{in}\, R'\, C\, \tau\, (\#)_{out}\, R'\, R''\, 1\, \tau' \rightarrow^{+} R'\, R''\, C\, \tau''\#\right]$$

ensures correctness of the all the remaining rows.

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

The negative query ψ will be a union of conjunctive queries describing possible errors in the encoding; this way, each word satisfying $\neg\psi$ will be a correct encoding of the tiling. To keep uniform notation, we present these queries as disjunctions of WPQs; the *in* and *out* nodes can be placed arbitrarily, as we only care for emptiness.

Possible coding errors are:

(1) symbols after $\tau_{\textit{fin}}\#$,
(2) # or tile identifier appearing in wrong position,
(3) two consecutive # symbols (empty row),
(4) the same row identifier used in two different rows,
(5) the same column identifier twice in one row.

To make sure that the first occurrence of $\tau_{\textit{fin}}\#$ ends the words we use

$$(\tau_{\textit{fin}})^{in}_{out} \rightarrow \# \rightarrow X\,.$$

The positive query φ ensures that row and next row are the same between two #'s, so to find # or tiles used as row identifiers it is enough to check first and second position

after each $\#$:

$$\bigvee_{\tau \in K \cup \{\#\}} (\#)^{in}_{out} \to \tau\,, \qquad \bigvee_{\tau \in K \cup \{\#\}} (\#)^{in}_{out} \to R \to \tau\,.$$

To find tiles or $\#$'s used as column identifiers, we rely on the fact that column identifier always precedes a tile:

$$\bigvee_{\tau \in K} \bigvee_{\tau' \in K \cup \{\#\}} (\tau')^{in}_{out} \to \tau\,.$$

The query for two consecutive $\#$ symbols is of course

$$(\#)^{in}_{out} \to \#\,.$$

To find the remaining errors—repeated column identifiers within a single row and repeated row identifiers in different rows—we need relation $\to^+$. For repeated row identifiers in different rows we use

$$(\#)^{in}_{out}\, R \to^+ \#\, R\,,$$

relying on the fact that query φ enforces that within each row, each position uses the same row identifier. For repeated column identifiers within a single row we apply

$$\bigvee_{\tau \in K} \bigvee_{\tau' \in K} R\, R'\, C\, \tau \to^+ R\, R''\, C\, \tau'\,,$$

using the fact that we have just excluded reusing row identifiers by different rows, and that tile identifiers are always placed at correct positions; query φ actually enforces that R' and R'' are equal, but there is no need to use that.

We need to show that a correct tiling exists if and only if $\varphi \wedge \neg\psi$ is satisfiable. Encoding of each correct tiling is a word satisfying $\varphi \wedge \neg\psi$. For the converse, let us take a word w satisfying $\varphi \wedge \neg\psi$. By the first subquery of ψ, the *out* node of φ is matched at the last position of w, and without loss of generality, we can assume that the *in* node of φ is mapped in the first position of w. That means that each position of the word w is visited by φ. Consequently, the word can be divided into rows (segments separated by $\#$'s), such that within each row

— the number of positions is divisible by 4,
— positions $1, 5, \ldots,$ have the same label: row identifier;
— positions $2, 6, \ldots,$ have the same label: next row identifier;
— positions $4, 8, \ldots,$ contain a tile, and subsequent tiles respect relation H.

By $\neg\psi$, tiles and $\#$ are not used in place of row, next row, or column identifier. By φ, the row identifier of each row (except the first), is equal to the next row identifier of the previous row. By $\neg\psi$, row and column identifiers are unique. Consequently, the fragment of subquery α connected with $\to^+$ is always matched in the next row. Since φ sets each first column identifier to 1, and copies the last column identifier from the previous row, the previous assertion implies the sequence of column identifiers in each row is the same (in particular, all rows have equal length), and the corresponding tiles satisfy relation V. □

PROPOSITION 5.2. *Over trees, both ranked and unranked, the containment problem of* L-Datalog$(\downarrow, \downarrow^+)$ *programs in* UCQ$(\downarrow, \downarrow^+)$ *queries is undecidable.*

PROOF. This follows directly from the proof of Proposition 5.1. It suffices to change the queries used in the word case by replacing $\to$ with $\uparrow$ and $\to^+$ with $\uparrow^+$. As there is

only one path up in the tree starting at any given node, the reduction behaves exactly like in the word case. □

We now adapt the above reduction to ranked trees, to prove the undecidability result from Theorem 3.3.

PROPOSITION 5.3. *Over ranked trees, containment of* LD-Datalog$(\downarrow,\downarrow^+)$ *programs in* UCQ$(\downarrow,\downarrow^+)$ *queries is undecidable.*

PROOF. We again modify slightly the reduction from Proposition 5.1. First, change the arrows from $\rightarrow$ to $\downarrow$. Next, let R be the maximal rank of the trees. The queries will use R additional symbols $\Sigma_R = \{\flat_1, \flat_2, \ldots, \flat_R\}$. We modify the positive query φ from Proposition 5.1 by adding under each node $R-1$ children with labels $\flat_1, \flat_2, \ldots, \flat_{R-1}$. For example, subquery $C'{\downarrow}\tau'$ of query β becomes the query shown in Figure 6.

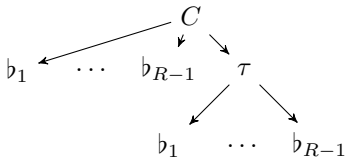


Fig. 6: $C{\downarrow}\tau$ modified

Additionally, in the query $\varphi_{\textit{last-row}}$, we add Rth child, with label $\flat_R$, below the last node. The negative query ψ additionally ensures there are no nodes below nodes labeled with elements of Σ_R. We add to ψ disjunct

$$\bigvee_{i=1}^{n} \flat_i \downarrow X\,.$$

This way the query φ has to be matched within a single path (the one without nodes from Σ_R) and thus the encoding of the tiling problem is correct. □

Finally, we prove Proposition 4.1, which we repeat below for convenience.

Proposition 4.1. *Over unranked trees containment of* Datalog$(\downarrow)$ *programs in* UCQ$(\downarrow)$ *queries is undecidable.*

PROOF. We show how to modify the encoding from Proposition 5.2 to work for Datalog$(\downarrow)$ programs (recall that the reduction in Proposition 5.2 was obtained by replacing $\rightarrow$ and $\rightarrow^+$ with $\uparrow$ and $\uparrow^+$ in the reduction from Proposition 5.1). The main difference is that now we cannot use the relation $\uparrow^+$. We shall show how to simulate it using recursion and non-linearity of the programs. We stress that our programs will not be equivalent: the simulation will require additional nodes in the underlying tree.

We shall work with datalog programs, rather than regular pattern queries. Let $\mathcal{P}$ be a datalog program equivalent to the positive query from the proof of Proposition 5.2. Let r be a rule of $\mathcal{P}$ that uses $\uparrow^+$ (formally, datalog programs use only $\downarrow$ and $\downarrow^+$, but we shall allow $\uparrow$ and $\uparrow^+$ as syntactic sugar enhancing readability). Since program $\mathcal{P}$ expresses an RTPQ whose patterns are not branching, rule r can be rewritten as

$$P(X_1) \leftarrow \quad C_1(\bar{X}), X_k{\uparrow^+}Y_1, C_2(\bar{Y}), C_{\sim}(\bar{X},\bar{Y}), \tag{2}$$

where $\bar{X} = X_1, \ldots, X_k$, $\bar{Y} = Y_1, \ldots, Y_\ell$ are disjoint tuples of variables, $C_1(\bar{X})$ and $C_2(\bar{Y})$ are conjunctions of atoms over $\bar{X}$ and $\bar{Y}$ respectively, $C_1(\bar{X})$ does not use $\uparrow^+$, and $C_{\sim}(\bar{X},\bar{Y})$ is a conjunction of atoms of the form $X_i \sim Y_j$.

We shall now eliminate $X_k{\uparrow^+}Y_1$ from r, obtaining a rule r', which uses a new intensional predicate U. Hence, rule r' may be non-linear if $C_1(\bar{X})$ contains an intensional

$$\mathcal{R} : R(X_1) \leftarrow \; a(X_1), X_1{\uparrow^+}Y, X_1 \sim Y$$

$$\mathcal{R}' : R(X_1) \leftarrow \; a(X_1), X_1{\uparrow}X_2', \\ path(X_1', X_2', X_3'), X_1 \sim X_1', U(X_1')$$

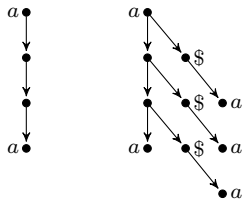


Fig. 7: Simulating the ancestor relation with recursion (irrelevant labels and definitions of predicates path and U are skipped). The intended model for the modified program $\mathcal{R}'$ (right) is obtained from a model for the original program $\mathcal{R}$ (left), by attaching additional subtrees along the path traversed by the ancestor relation.

atom. Rule r' is given as

$$P(X_1) \leftarrow \; C_1(\bar{X}), X_k{\uparrow}X'_{k+2}, \\ \mathrm{path}(X'_1, \ldots, X'_{k+2}), \bigwedge_{i=1}^{k} X_i \sim X'_i, U(X'_1),$$

where $\mathrm{path}(X'_1, \ldots, X'_{k+2}) = \bigwedge_{i=1}^{k+1} X'_i {\uparrow} X'_{i+1}, \$(X'_{k+1})$ and $\$$ is a fresh label. The idea is that r' matches $C_1(\bar{X})$, but instead of matching $C_2(\bar{Y})$, it requires a copy of $\bar{X}$ with the same labels in a subtree (single path formed by $X'_1, \ldots, X'_k$) attached under the parent X'_{k+2} of X_k, additionally separated with a $\$$-labelled node X'_{k+1}. Originally in r we had $X_k{\uparrow^+}Y_1$; here, intuitively, X'_{k+2} is the first candidate for Y_1. This copying will be repeated by U, until $C_2(\bar{Y})$ can be matched (see Figure 7); the reason for copying is that while matching $C_2(\bar{Y})$ we need to satisfy the label equalities in $C_\sim(\bar{X}, \bar{Y})$. We implement it as follows:

$$U(X_1) \leftarrow \; \mathrm{path}(X_1, \ldots, X_{k+2}), X_{k+2}{\uparrow}X'_{k+2}, \\ \mathrm{path}(X'_1, \ldots, X'_{k+2}), \bigwedge_{i=1}^{k} X_i \sim X'_i, U(X'_1)\,, \\ U(X_1) \leftarrow \; \mathrm{path}(X_1, \ldots, X_{k+1}, Y_1), C_2(\bar{Y}), C_\sim(\bar{X}, \bar{Y})\,.$$

Thus we have removed one occurrence of $\uparrow^+$. Now, if $C_2(\bar{Y})$ contains further occurrences of $\uparrow^+$, simply repeat the whole procedure for the second rule for U, etc.

This way we change the rules of the positive program $\mathcal{P}$. We cannot use the same technique for the negative program $\mathcal{Q}$ for two reasons. First, it uses recursion and we would like to keep $\mathcal{Q}$ non-recursive to get undecidability of containment in UCQs. More importantly, it requires additional nodes in the tree: if those nodes are not there, the tree does not satisfy the modified $\mathcal{Q}$.

In order to deal properly with relation $\uparrow^+$ in program $\mathcal{Q}$, we need to recall the details of the reduction from the proof of Proposition 5.2. Recall that we encoded each tile as a tuple (n, n', m, t), where n is the the row identifier, n' is the next row identifier, m is the column identifier, and t is a tile. Consecutive rows were separated by $\#$. For the correctness of the reduction it was essential that row identifiers and column identifiers be unique. The negative program $\mathcal{Q}$ needed relation $\uparrow^+$ only to detect two columns with the same identifier within a single row of the tiling, and two rows with the same identifier. Here, we take care of these two kinds of errors in a different way.

Suppose there is a tiling with k rows and l columns. Let $r_1, \ldots, r_k$ be row identifiers and let $c_1, \ldots, c_l$ be the column identifiers. We shall modify program $\mathcal{P}$ to enforce that

(for all nodes that are part of the encoding) each r_i-labelled node has $i-1$ additional children with labels $r_1, \dots r_{i-1}$, and each c_j-labelled node has $j-1$ children with labels $c_1, \dots c_{j-1}$. To ensure the uniqueness of row and column identifiers, we add to $\mathcal{Q}$ a rule that does not allow nodes to have a child with the same label.

Whenever a variable X in rule r of $\mathcal{P}$ refers to a node representing a column identifier, we add an additional atom $E(X)$ to r. Predicate E is defined as

$$\begin{aligned} E(X) \leftarrow &\quad X \uparrow^4 Y \uparrow Z,\, \tau(Z),\, Y \downarrow X',\, X \sim X',\, E'(X')\,, \\ E(X) \leftarrow &\quad X \uparrow^2 Y, \#(Y)\,, \end{aligned}$$

$$\begin{aligned} E'(X) \leftarrow &\quad X \uparrow^5 Y \uparrow Z,\, \tau(Z),\, Y \downarrow X',\, X \sim X',\, E'(X')\,, \\ E'(X) \leftarrow &\quad X \uparrow^3 Y, \#(Y)\,, \end{aligned}$$

where τ ranges over the set of tiles, expressions like $X \uparrow^4 Y \uparrow Z$ are shorthands for $X \uparrow^4 Y$, $Y \uparrow Z$, and $\uparrow^k$ stands for the k-fold composition of relation $\uparrow$, clearly expressible with a CQ, which can be inlined in the rule. That is, we copy the label of X (column identifier) into the label of a child of each ancestor node representing a column, until we see $\#$, indicating the end of the row. In the first rule we check if the parent Z of Y is labeled with a tile, because we do not want to apply this rule when the upward path is a proper encoding and the row changes. Since in the first step we go up one level less then in the following steps, we need an auxiliary predicate E'.

Similarly, whenever variable X refers to a node representing a row identifier, we add atom $F(X)$, with predicate F defined as

$$\begin{aligned} F(X) \leftarrow &\quad X \uparrow^4 Y \uparrow^3 Z, \tau(Z), Y \downarrow X', X \sim X', F'(X')\,, \\ F(X) \leftarrow &\quad X \uparrow^5 Y \uparrow^3 Z, \tau(Z), Y \downarrow X', X \sim X', F'(X')\,, \\ F(X) \leftarrow &\quad X \uparrow^3 Y \uparrow Z,\, \tau_{\mathit{fin}}(Y), \#(Z)\,, \end{aligned}$$

$$\begin{aligned} F'(X) \leftarrow &\quad X \uparrow^5 Y \uparrow^3 Z, \tau(Z), Y \downarrow X', X \sim X', F'(X')\,, \\ F'(X) \leftarrow &\quad X \uparrow^6 Y \uparrow^3 Z, \tau(Z), Y \downarrow X', X \sim X', F'(X')\,, \\ F'(X) \leftarrow &\quad X \uparrow^4 Y \uparrow Z,\, \tau_{\mathit{fin}}(Y), \#(Z)\,. \end{aligned}$$

The middle rule for F and F' is needed to jump over $\#$-labelled nodes: unlike for column identifiers, we want to continue copying until we see the final tile τ_{fin} followed by $\#$, which indicates the end of the last row.

The negative query $\mathcal{Q}$ is obtained by taking the union of all the $\uparrow^+$-free CQs forming the negative query in the proof of Proposition 5.2, together with the following simple CQ, which prevents a node and its child from having the same label:

$$\exists X\ \exists Y\ X{\downarrow}Y \wedge X \sim Y\,.$$

Combined with the mechanism for copying labels representing row and column identifiers, described above, this ensures that row and column identifiers are unique. Note that in the simulation of $\uparrow^+$ in program $\mathcal{P}$ we separated the copied nodes with \$ precisely to avoid satisfying this CQ unintentionally. □

6. LOWER BOUNDS

In this section we present the hardness arguments, organized into four propositions. The first two use RTPQs and their word variant, RWPQs (see Section 5); by Proposition 2.6 the established bounds carry over to the corresponding fragments of datalog.

We begin with a simple reduction proving the PSPACE lower bound for satisfiability over words, claimed in Proposition 3.1 (Section 3).

PROPOSITION 6.1. *Satisfiability for* RWPQ($\rightarrow$) *is* PSPACE*-hard.*

PROOF. In the proof we use the RWPQ notation for programs on words introduced in Section 5. To prove hardness, we reduce the space-bounded halting problem known to be PSPACE-complete. More precisely, let n be a number given in unary and let $\mathcal{M}$ be a Turing machine using not more than n tape cells. We construct an RWPQ of size polynomial in $|\mathcal{M}|$ and n that is satisfiable if and only if $\mathcal{M}$ accepts the empty word. In this reduction it is not important whether nodes have labels taken from a finite or infinite alphabet: one can restrict to labels explicitly mentioned in the RWPQ.

Assume that Γ is the tape alphabet of $\mathcal{M}$ containing the blank symbol $\bot$, Q is the set of states containing the initial state q_{ini} and the accepting state q_{fin}, and δ is the transition relation. Without loss of generality we can assume that the fixed alphabet Σ contains Q and two disjoint copies of Γ, denoted by Γ and $\widehat{\Gamma}$. The symbols from $\widehat{\Gamma}$ are decorated with $\widehat{\ }$; they will be used to mark the head's position on the tape: $\widehat{a}$ means that the machine's head is over this occurrence of symbol a.

The basic building block of our RWPQ is a WPQ for one transition rule and position on the tape (we skip most $\rightarrow$ symbols for readability):

$$(q_1)^{in} X_1 X_2 \dots \mathit{Head} \dots X_n \rightarrow$$
$$(q_2)_{out} X_1 X_2 \dots \mathit{Head}' \dots X_n$$

where q_1, q_2 are states of M, Head and Head' are 3-symbol descriptions of the tape contents around the machine's head. Specific content of Head will be determined by the type of transition (whether the head moves right, left or stays in the same position). Note that the *out* node is located on the state symbol in the second configuration. This way, when we apply Kleene star to our pattern, consecutive patterns will overlap, ensuring that the run is correct.

For different head movements, patterns will be different but very similar. For example, for head over i-th tape cell, transition t saying "in state q_1, seeing letter a, change state to q_2, write c, and move the head to the right", and letter b in the next tape cell, we use WPQ

$$\varphi_{i,t,b} = (q_1)^{in} X_1 X_2 \dots X_{i-1}\, \widehat{a}\, b\, X_{i+2} \dots X_n$$
$$\rightarrow (q_2)_{out} X_1 X_2 \dots X_{i-1}\, c\, \widehat{b}\, X_{i+2} \dots X_n\,.$$

The variables X_i repeating in both configurations ensure that the content of the tape does not change. Symbol $\widehat{a}$ is changed to c, and b becomes $\widehat{b}$ to mark the new position of the head. Additionally, there must be separate patterns for head positions in the first and the last tape symbol.

The final RWPQ is defined as

$$\varphi_{ini} \cdot \left(\bigvee_{i=1}^{n} \bigvee_{t \in \delta} \bigvee_{b \in \Gamma} \varphi_{i,t,b} \right)^{*} \cdot \varphi_{fin}$$

where

$$\varphi_{ini} = (q_{ini})^{in}_{out} \widehat{\bot} \bot \dots \bot\,, \qquad \varphi_{fin} = (q_{fin})^{in}_{out} X_1 X_2 \dots X_n$$

describe initial and final configurations, respectively.

Take a word satisfying this RWPQ and trim the infix and suffix that is not used in the matching. The remaining word clearly encodes a correct accepting run of $\mathcal{M}$, using

at most n tape cells. Conversely, each run using at most n tape cells is encoded as a word satisfying this RWPQ.

One remaining issue is the size of the constructed query. Let $\Sigma_0 = Q \cup \Gamma \cup \widehat{\Gamma}$ be the set of all labels explicitly mentioned in the RWPQ. Patterns φ_{ini} and φ_{fin} have size $n+1$, and each pattern $\varphi_{d,i,b}$ has size $2(n+1)$. Since there are at most $n \cdot |\delta| \cdot |\Sigma_0|$ patterns in the disjunction, the total size of the query is polynomial in $|\mathcal{M}|$ and n. □

We now move to the EXPSPACE lower bound of Theorem 4.2 (Section 4.1). We show hardness already for containment in non-recursive queries, that is, UCQs. In terms of patterns, UCQs are unions (or disjunctions) of TPQs.

PROPOSITION 6.2. *Over unranked trees containment of* $\mathsf{RTPQ}(\downarrow, \downarrow^+)$ *queries in unions of* $\mathsf{TPQ}(\downarrow, \downarrow^+)$ *queries is* EXPSPACE*-hard.*

PROOF. We give a reduction from the following variant of the tiling problem: Given

— a number n (in unary),
— a finite set of tiles K,
— a horizontal correctness relation $H \subseteq K \times K$,
— a vertical correctness relation $V \subseteq K \times K$,
— an initial tile τ_{ini},
— a final tile τ_{fin},

decide if there exists a natural number m such that there is a tiling of the $m \times 2^n$ grid that has τ_{ini} as the top left tile, i.e., the tile in position $(1,1)$, τ_{fin} as the bottom right tile, i.e., the tile in position $(m, 2^n)$, and all pairs of adjacent tiles satisfy the appropriate relation, H or V. It is easy to prove that this is an EXPSPACE-complete problem: such a tiling can be seen as a run of a Turing machine using 2^n tape cells.

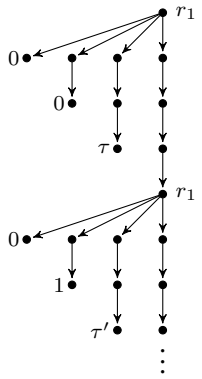


Fig. 8: Fragment of a model encoding a tiling, showing the first two tiles, τ and τ' ($n = 2$, irrelevant labels are skipped). The first row is identified by the label r_1, column identifiers are encoded in binary.

We shall construct query $\varphi \in \mathsf{RTPQ}(\downarrow, \downarrow^+)$ and query ψ, a union of $\mathsf{TPQ}(\downarrow, \downarrow^+)$ queries, such that $\varphi \wedge \neg\psi$ is satisfiable if and only if the tiling exists. Query φ will encode the tiling and the query ψ will ensure its correctness; the encoding is similar to the one used to prove Proposition 5.1. There, we worked with words and we could easily force consistent ordering of column identifiers. Now we work with trees and this is no longer

possible; instead, we use explicit binary encoding of column identifiers. The tiling is thus encoded as sequence of segments of the form $r\, b_1\, b_2\, \ldots\, b_n\, \tau$ where r is the row identifier, $b_1\, b_2\, \ldots\, b_n$ encode column identifier in binary, and τ is a tile symbol (see Figure 8). Without loss of generality we can assume that the fixed infinite labeling alphabet Σ contains set K, as as well as labels 0 and 1 used to encode column identifiers.

We begin with query φ. Recall that we write $\downarrow^i$ for the i-fold composition of relation $\downarrow$, which can be expressed with a $\downarrow$-pattern (using fresh variables for intermediate nodes).

To check horizontal and vertical correctness relations we use expression

$$\alpha = \bigvee_{(\tau,\tau')\in H} \bigvee_{(\tau,\tau'')\in V} \underbrace{(R)^{in}_{out}\downarrow B_1\downarrow \ldots \downarrow B_n\downarrow\tau}_{\text{current tile}} \downarrow \underbrace{R\downarrow^{n+1}\tau'}_{\text{next tile}} \downarrow^+ \underbrace{R\downarrow 1\downarrow \ldots\downarrow 1}_{\text{last tile in row}} \downarrow X \downarrow R'\downarrow^+ \underbrace{R'\downarrow B_1\downarrow \ldots\downarrow B_n\downarrow\tau''}_{\text{tile below current}}$$

(the last tile in the row is visited only for navigational purposes). We need special variants of this expression, α_{first}, α_{penul} and α_{last}, for the cases where the current tile is the first, penultimate and last tile in a row. They cover edge cases: the tile below current starting immediately after X, next tile equal to the last tile, and no next tile. For example, α_{last} is defined as follows:

$$\alpha_{\mathit{last}} = \bigvee_{(\tau,\tau'')\in V} \underbrace{(R)^{in}_{out}\downarrow 1\downarrow \ldots\downarrow 1\downarrow\tau}_{\text{current (last) tile}} \downarrow R'\downarrow^+ \underbrace{R'\downarrow 1\downarrow \ldots\downarrow 1\downarrow\tau''}_{\text{tile below current}} .$$

Expressions α_{first} and α_{last} additionally check that the current tile is indeed first or last, by using 0s or 1s instead of variables B_i.

To combine these expressions we need a way to move to the next tile, where we touch the most delicate aspect of the reduction. Making the step requires reading the current column identifier and the next column identifier; that is, we must read each column identifier twice. Given that we are only allowed to go down the tree, we cannot ensure that the two reads go down the same branch; we can only control depth in the tree. The idea is to ensure that each node within one generation of descendants of the current tile encodes the same bit of the column identifier: first bit in the first generation, second bit in the second generation, etc. The next tile is stored in generation $(n+1)$. The integrity of this encoding will be ensured by the negative query ψ. Keeping that in mind, we check the whole row with expression φ_{row}, defined as

$$\varphi_{\mathit{row}} = \big(\alpha_{\mathit{first}}\,\beta\,\gamma\big)\cdot\big(\alpha\,\beta\,\gamma\big)^*\cdot\big(\alpha_{\mathit{penul}}\,\beta\,\gamma\big)\cdot\big(\alpha_{\mathit{last}}\,\beta\big)\,,$$

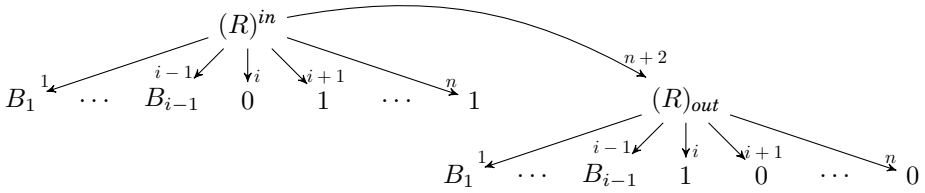
where expression β ensures that each generation (from 0 to n) contains 0 or 1,

$$\beta = \big((R)^{in}_{out}\downarrow^1 0 + (R)^{in}_{out}\downarrow^1 1\big)\cdot\big((R)^{in}_{out}\downarrow^2 0 + (R)^{in}_{out}\downarrow^2 1\big)\cdot\cdots\cdot\big((R)^{in}_{out}\downarrow^n 0 + (R)^{in}_{out}\downarrow^n 1\big)\,,$$

and expression γ moves to the next tile and increments the column identifier,

$$\gamma = \bigvee_{i=1}^{n} \gamma_i$$

with pattern γ_i defined as



The query φ can be now defined as

$$\varphi = \varphi_{\textit{first-tile}} \cdot (\varphi_{\textit{row}}\ \delta_{\textit{next}})^* \cdot \varphi_{\textit{last-row}} \cdot \varphi_{\textit{last-tile}}\ ,$$

where expressions $\varphi_{\textit{first-tile}}$ and $\varphi_{\textit{last-tile}}$ enforce correct initial and final tile,

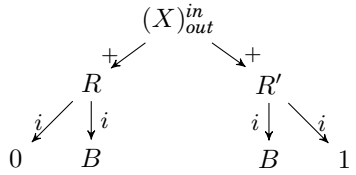
$$\varphi_{\textit{first-tile}} = (R)^{\textit{in}}_{\textit{out}} \downarrow^{n+1} \tau_{\textit{ini}}\ , \qquad \varphi_{\textit{last-tile}} = (R)^{\textit{in}}_{\textit{out}} \downarrow^{n+1} \tau_{\textit{fin}}\ ,$$

$\delta_{\textit{next}}$ moves to the next row,

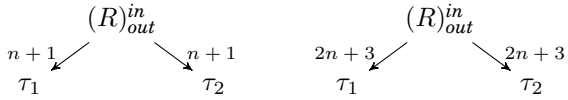
$$\delta_{\textit{next}} = (R)^{\textit{in}} \downarrow^{n+2} (R')_{\textit{out}}\ ,$$

and $\varphi_{\textit{last-row}}$ is a variant of $\varphi_{\textit{row}}$ in which references to the next row are eliminated.

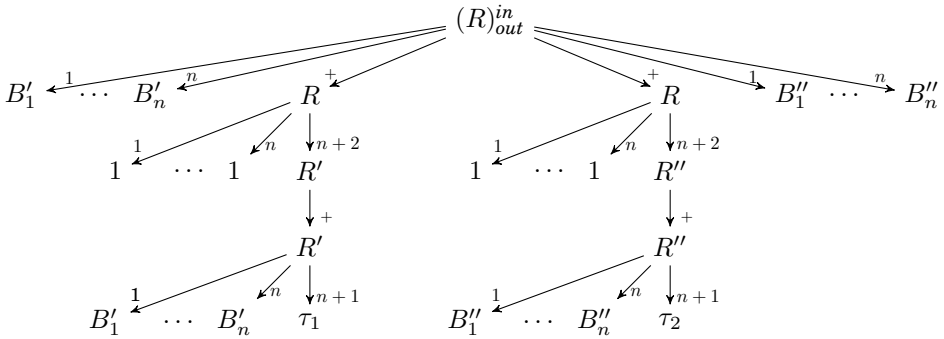
We now move to query ψ, which detects inconsistencies in the encoding. We begin with column identifiers. Already query β ensures that each generation of descendants of a tile contains 0 or 1; but it must not contain both. Moreover, generations containing 0 and those containing 1 may contain other values too: they must not be mixed. Both conditions are ensured by the following disjuncts of ψ, with $1 \leq i \leq n$, which check that no value occurs in an ith generation containing 0 and an ith generation containing 1:



Next we deal with tiles. To ensure unique current and next tile we use disjuncts



for all tiles $\tau_1 \neq \tau_2$. For the tile below the current tile we need a more complicated expression. For all tiles $\tau_1 \neq \tau_2$ we add disjunct



which finds two copies of the last tile of a row R, moves down to two instances of the next row, and finds different tiles τ_1 and τ_2 in columns whose identifiers, encoded by $B'_1 \dots B'_n$ and $B''_1 \dots B''_n$, are equal. We cannot test equality directly, because the encodings need not be identical. Indeed, to address the tile below current, expression α uses arbitrary representatives $B_1, B_2, \dots, B_n$ (not necessarily 0s or 1s) of the bits of the current tile's column identifier. In fact, there may be no 0s and 1s at all in the generations encoding the column identifier of the tile below current. Because of that, instead of comparing all possible instances of the tile below current with each other, the expression compares all of them with the unique instance on the main branch visited by query φ. For this instance, expression β does guarantee 0 or 1 in each generation. To test equality of column identifiers, we match $B''_1 \dots B''_n$ to these 0s and 1s, and check if they occur in pairs with $B'_1 \dots B'_n$ in the generations encoding the current tile's column identifier. This is correct by the first collection of disjuncts of expression ψ. □

Next, we establish the 2-EXPTIME lower bound of Theorem 4.2 (Section 4.1), and the first strengthening of the one from Theorem 3.3 (Section 3).

PROPOSITION 6.3. *Containment for* D-Datalog(↓) *is 2-*EXPTIME*-hard over trees and ranked trees.*

PROOF. Consider the following problem: given a natural number n in unary and an alternating Turing machine $\mathcal{M}$, decide if $\mathcal{M}$ accepts the empty word using at most 2^n tape cells. This problem is AEXPSPACE-complete and AEXPSPACE $=$ 2-EXPTIME. We construct a pair of programs $\mathcal{P}, \mathcal{Q}$ such that $\mathcal{P} \wedge \neg\mathcal{Q}$ is satisfiable iff $\mathcal{M}$ accepts 1^n.

Let Q be the set of states of machine $\mathcal{M}$, containing the initial state q_{ini} and the accepting state q_{fin}. Without loss of generality we assume that state q_{ini} is used only once in each run (no transitions lead to it) and that each configuration with a universal state has exactly two successor configurations: for each *universal* state q and each letter a, machine $\mathcal{M}$ has exactly two transitions. Each transition is of the form (q, a, q', b, d), where b is the letter to be written in the current cell and $d \in \{L, S, R\}$ specifies where the machine's head moves: left, stay, or right.

The run of $\mathcal{M}$ is a binary tree. Each node of the tree is labelled with a pair (q, T) where q is a state and T is the content of the tape with one distinguished cell containing the head. If q is universal, the node has two children; if q is existential, it has one child. In the root the state must be q_{ini}, and in leaves only the accepting state q_{fin} is allowed.

Let us describe how we encode the runs. Each configuration (q, T) is represented by a *state node* u labeled with state q and two or three subtrees, depending on whether q is existential or universal. The first subtree, called *tape tree*, encodes tape content T (described later). The remaining subtrees, one or two, encode successor configurations together with the transitions generating them. Figure 9 presents a fragment of the encoding of a run: from the initial state q_{ini} existentially chosen transition t_1 moves to state q_1, then universally chosen transitions t_2 and t_3 move to states q_2 and q_3, etc.; tape trees T_i encode the corresponding tape contents.

This encoding requires trees of rank at least three, but it is easy to adjust it to binary trees, by introducing an additional dummy node for universal states. For the sake of readability we shall continue the reduction with this encoding.

Tape trees are full binary trees of height n. Such tree has 2^n leaves so the content of every tape cell can be stored in a node directly below a leaf of this subtree. We use the set of labels $\{(0,0)\} \cup \big\{(0,i),(1,i) \;\big|\; 1 \le i \le n\big\}$. The root of each tape tree has label $(0,0)$ and each node at level $i-1$ has two children, labelled with $(0,i)$ and $(1,i)$; slightly informally, we can think of them as the left and right child. Below the nodes on the n-th level, labelled with $(0,n)$ or $(1,n)$, we encode the content of the cells. For that we use labels from the set $\Gamma \times \{\bot, \top\}$, where Γ is the tape alphabet of $\mathcal{M}$. With $\top$ we mark

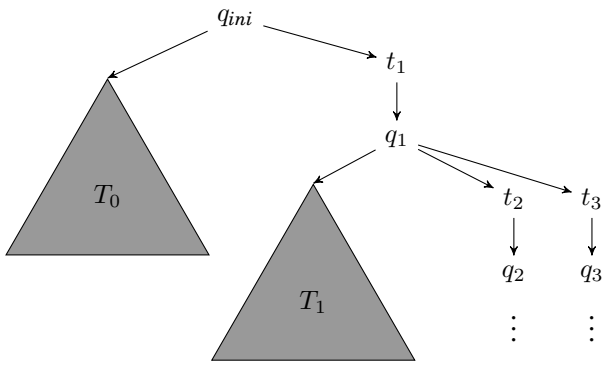


Fig. 9: Encoding of a run.

the unique position of the head of $\mathcal{M}$. Consider a branch of the tape tree; the labels of the nodes on this branch are $(0,0),(z_1,1),\ldots(z_n,n),b$ with $z_i \in \{0,1\}$. Then, b is the content of the cell whose number is encoded in binary by the sequence $z_1 \ldots z_n$.

We are now ready to describe the rules of program $\mathcal{P}$. Recall that we use expression $X_1 \downarrow X_2 \downarrow \ldots \downarrow X_k$ as shorthand for $X_1 \downarrow X_2, X_2 \downarrow X_3, \ldots, X_{k-1} \downarrow X_k$. The goal rule is

$$\underline{P(X)} \leftarrow \quad q_{ini}(X), run(X)\,.$$

For each state q of machine $\mathcal{M}$,

— if q is existential, for each transition $t = (q, a, q_1, b, d)$, with $d \in \{L, S, R\}$ we add rule

$$\begin{aligned} run(X) \leftarrow \quad & q(X), X \downarrow X', (0,0)(X'), conf(X'), \\ & X \downarrow Y_1 \downarrow Z_1, t(Y_1), q_1(Z_1), run(Z_1)\,; \end{aligned}$$

— if q is universal, for each pair of transitions $t_i = (q, a, q_i, b_i, d_i)$, $i = 1, 2$, we add rule

$$\begin{aligned} run(X) \leftarrow \quad & q(X), X \downarrow X', (0,0)(X'), conf(X'), \\ & X \downarrow Y_1 \downarrow Z_1, t_1(Y_1), q_1(Z_1), run(Z_1), \\ & X \downarrow Y_2 \downarrow Z_2, t_2(Y_2), q_2(Z_2), run(Z_2)\,. \end{aligned}$$

Additionally, we include a rule for the accepting state

$$run(X) \leftarrow \quad q_{fin}(X), X \downarrow X', (0,0)(X'), conf(X')\,.$$

Notice that these rules ensure only the correctness of states in successor configurations. Predicate $conf$ enforces a tape tree under each state node; correct evolution of tape content will be verified later by the negative program $\mathcal{Q}$. The rules for $conf$ are

$$\begin{aligned} conf(X) \leftarrow \quad & (x, i-1)(X), \\ & X \downarrow Y, (0,i)(Y), conf(Y), \\ & X \downarrow Z, (1,i)(Z), conf(Z) \\ conf(X) \leftarrow \quad & (x, n)(X), \\ & X \downarrow Y, a(Y) \end{aligned}$$

for $i = 1, \ldots, n$, $x \in \{0,1\}$ such that $(x, i-1) \neq (1,0)$, and $a \in \Gamma \times \{\bot, \top\}$.

Of course, a model for $\mathcal{P}$ doest not need to be a correct run of $\mathcal{M}$. To ensure correctness, we need to detect the following errors:

(1) two different contents in the same tape cell;
(2) incorrect initial tape content;

(3) inconsistent tape content in consecutive configurations.

This will be done by the negative program $\mathcal{Q}$ by means of a disjunction of non-recursive queries, each detecting some kind of error. Let us start with (1). In general the models can be unranked trees and the program $\mathcal{P}$ cannot ensure that nodes do not have children with the same labels. Thus, each cell might be encoded by many paths. To ensure that all these paths encode the same cell content, we add rule

$$\begin{aligned}\mathcal{Q}(X) \leftarrow \quad & (0,0)(X),\\ & X \downarrow Y_1 \downarrow Y_2 \downarrow \ldots \downarrow Y_n \downarrow Z,\ c(Z),\\ & X \downarrow Y_1' \downarrow Y_2' \downarrow \ldots \downarrow Y_n' \downarrow Z', c'(Z'),\\ & Y_1 \sim Y_1',\ Y_2 \sim Y_2',\ \ldots,\ Y_n \sim Y_n'\end{aligned}$$

for each pair of different tape contents, $c \neq c'$.

To check (2) we detect nonempty cells and head in any cell but the first one

$$\begin{aligned}\mathcal{Q}(X) \leftarrow \quad & X \downarrow Y_0 \downarrow Y_1 \downarrow \ldots \downarrow Y_n \downarrow Z,\ q_{ini}(X), (0,1)(Y_1), \ldots, (0,n)(Y_n), a(Z)\\ \mathcal{Q}(X) \leftarrow \quad & X \downarrow Y_0 \downarrow Y_1 \downarrow \ldots \downarrow Y_n \downarrow Z,\ q_{ini}(X), (1,i)(Y_i), b(Z)\end{aligned}$$

for $i = 1, 2, \ldots, n$ and tape contents $a \neq (\flat, \top)$, $b \neq (\flat, \bot)$, where $\flat \in \Gamma$ stands for blank.

Checking (3) is more involved. To verify that the tape content evolves correctly, it suffices to compare triples of consecutive cells in consecutive configurations. This can be done with rules

$$\begin{aligned}Q(X) \leftarrow \quad & X \downarrow X_1 \downarrow X_2, t(X_1),\\ & X \downarrow Z_0,\ \mathit{triple}(Z_0, Z_1, \ldots, Z_n, Y_1, Y_2, Y_3),\ a(Y_1),\ b(Y_2),\ c(Y_3),\\ & X_2 \downarrow Z_0', \mathit{triple}(Z_0', Z_1', \ldots, Z_n', Y_1', Y_2', Y_3'), a'(Y_1'), b'(Y_2'), c'(Y_3'),\\ & Z_0 \sim Z_0',\ Z_1 \sim Z_1',\ \ldots\ ,\ Z_n \sim Z_n',\end{aligned}$$

for all $a, b, c, a', b', c' \in \Gamma \times \{\bot, \top\}$ such that consecutive symbols a, b, c cannot change to a', b', c' when machine $\mathcal{M}$ makes transition t; but we need to define predicate $\mathit{triple}(Z_0, Z_1, \ldots, Z_n, Y_1, Y_2, Y_3)$ which selects three consecutive cells Y_1, Y_2, Y_3 in the tape tree rooted at Z_0, with $Z_1, \ldots, Z_n$ forming the path to Y_1. It can be defined by the following rules for $i = 1, 2, \ldots, n-1$:

$$\begin{aligned}\mathit{triple}(Z_0, Z_1 \ldots, Z_n, Y_1, Y_2, Y_3) \leftarrow \quad & Z_0 \downarrow \ldots \downarrow Z_{i-1},\\ & Z_{i-1} \downarrow Z_i \downarrow \ldots \downarrow Z_n \downarrow Y_1,\ (0,i)(Z_i), (1,i+1)(Z_{i+1}), \ldots, (1,n-1)(Z_{n-1}), (0,n)(Z_n),\\ & Z_{i-1} \downarrow Z_i' \downarrow \ldots \downarrow Z_n' \downarrow Y_2,\ (0,i)(Z_i'), (1,i+1)(Z_{i+1}'), \ldots, (1,n-1)(Z_{n-1}'), (1,n)(Z_n'),\\ & Z_{i-1} \downarrow Z_i'' \downarrow \ldots \downarrow Z_n'' \downarrow Y_3, (1,i)(Z_i''), (0,i+1)(Z_{i+1}''), \ldots, (0,n-1)(Z_{n-1}''), (0,n)(Z_n''),\end{aligned}$$

$$\begin{aligned}\mathit{triple}(Z_0, Z_1, \ldots, Z_n, Y_1, Y_2, Y_3) \leftarrow \quad & Z_0 \downarrow \ldots \downarrow Z_{i-1},\\ & Z_{i-1} \downarrow Z_i \downarrow \ldots \downarrow Z_n \downarrow Y_1,\ (0,i)(Z_i), (1,i+1)(Z_{i+1}), \ldots, (1,n-1)(Z_{n-1}), (1,n)(Z_n),\\ & Z_{i-1} \downarrow Z_i' \downarrow \ldots \downarrow Z_n' \downarrow Y_2,\ (1,i)(Z_i'), (0,i+1)(Z_{i+1}'), \ldots, (0,n-1)(Z_{n-1}'), (0,n)(Z_n'),\\ & Z_{i-1} \downarrow Z_i'' \downarrow \ldots \downarrow Z_n'' \downarrow Y_3, (1,i)(Z_i''), (0,i+1)(Z_{i+1}''), \ldots, (0,n-1)(Z_{n-1}''), (1,n)(Z_n''),\end{aligned}$$

corresponding to two types of triples of consecutive binary numbers:

$$\begin{aligned}& z_1 \ldots z_{i-1} 011 \ldots 110, && z_1 \ldots z_{i-1} 011 \ldots 111,\\ & z_1 \ldots z_{i-1} 011 \ldots 111, && z_1 \ldots z_{i-1} 100 \ldots 000,\\ & z_1 \ldots z_{i-1} 100 \ldots 000, && z_1 \ldots z_{i-1} 100 \ldots 001.\end{aligned}$$

As it is written, program $\mathcal{Q}$ is not a UCQ, because it uses nested intentional predicate $triple$. Actually, it is not even monadic, because predicate $triple$ is not unary. However, being used only twice, predicate $triple$ can be in-lined without exponential blow-up. □

Finally, we show the 2-EXPTIME lower bound from Theorem 4.6 (Section 4.2), and the second strengthening of the one from Theorem 3.3 (Section 3).

PROPOSITION 6.4. *Containment for* L-Datalog(↓) *is 2-*EXPTIME*-hard over trees and ranked trees.*

PROOF. For L-Datalog(↓) we reduce the same problem and use the same encoding as in Proposition 6.3. This time, we can only use linear rules, but the programs do not have to be downward: they can walk up and down the tree. To enhance readability of the program we allow predicate ↑ as syntactic sugar. The tree encoding the run of $\mathcal{M}$ will be traversed by program $\mathcal{P}$ in the prefix order. Recall that we assume that the initial state q_{ini} is only used once in each run (no transitions lead to it), so the program knows when to stop after traversing the whole tree.

The goal rule remains unchanged,

$$\underline{P(X)} \leftarrow \quad q_{ini}(X), \mathit{run}(X).$$

Predicate *run* over a node with label q ensures a child with label $(0,0)$ that is the root of a tape tree, i.e., a full binary tree of depth n storing tape content of machine $\mathcal{M}$:

$$\begin{aligned}
\mathit{run}(X) \leftarrow \quad & q(X), X \downarrow Y, (0,0)(Y), \mathit{conf}(Y) \\
& \text{for all } q \in Q, \\
\mathit{conf}(X) \leftarrow \quad & (x,i)(X), X \downarrow Y, (0,i+1)(Y), \mathit{conf}(Y) \\
& \text{for all } i = 0, \ldots, n-1 \text{ and } x = 0, 1, \\
\mathit{conf}(X) \leftarrow \quad & (x,n)(X), X \downarrow Y, b(Y), \mathit{upConf}(X) \\
& \text{for all } b \in \Gamma \times \{\bot, \top\} \text{ and } x = 0, 1, \\
\mathit{upConf}(X) \leftarrow \quad & (0,i)(X), X \uparrow Y \downarrow Z, (1,i)(Z), \mathit{conf}(Z) \\
\mathit{upConf}(X) \leftarrow \quad & (1,i)(X), X \uparrow Y, \mathit{upConf}(Y) \\
& \text{for all } i = 1, \ldots, n\,.
\end{aligned}$$

After coming back to the node with label $(0,0)$, program $\mathcal{P}$ continues checking the run of $\mathcal{M}$ by moving to the next configuration:

$$\mathit{upConf}(X) \leftarrow \quad (0,0)(X), X \uparrow Y \downarrow Z' \downarrow Y',\ q(Y), t(Z'), s(Y'), \mathit{run}(Y')\,,$$

where t ranges over transitions from *existential* state q to state s, and the first of each pair of transitions over the same letter from *universal* state q to state s (we assume an arbitrary fixed order within these pairs; the second transition of the pair will be checked after traversing the subtree rooted at the s node). If the current state is accepting, program $\mathcal{P}$ backtracks to check the rest of the tree

$$\mathit{upConf}(X) \leftarrow \quad (0,0)(X), X \uparrow Y,\ q_{fin}(Y), \mathit{upRun}(Y)\,,$$

which is done with the help of predicate *upRun*, similar to *upConf*:

$$\begin{aligned}
\mathit{upRun}(Y) \leftarrow \quad & Y \uparrow Z \uparrow Y',\ q(Y'), t_1(Z), s_1(Y), \\
& Y' \downarrow Z'' \downarrow Y'', t_2(Z'), s_2(Y''), \mathit{run}(Y'')\,, \\
\mathit{upRun}(Y) \leftarrow \quad & Y \uparrow Z \uparrow Y',\ q(Y'), t(Z), s(Y), \\
& \mathit{upRun}(Y')\,, \\
\mathit{upRun}(Y) \leftarrow \quad & Y \uparrow Z,\ q_{ini}(Z)\,,
\end{aligned}$$

where t_1, t_2 range over pairs of transitions from *universal* state q over the same letter, leading to states s_1, s_2; t ranges over transitions from *existential* state q to state s and the second of each pair of transitions over the same letter from *universal* state q to state s; the program terminates when it returns to the (unique) node labelled with q_{ini}.

The negative program $\mathcal{Q}$ needs one additional rule

$$Q(X) \leftarrow \quad Y \downarrow X, q_{ini}(X)\,,$$

which ensures that label q_{ini} can appear only in the actual root of the tree. Otherwise, rules of $\mathcal{Q}$ may remain as in Proposition 6.3, since every UCQ is a linear program. □

7. CLOSING REMARKS

The containment problem for connected monadic datalog on trees over infinite alphabet is known to be undecidable [Abiteboul et al. 2013]. We investigated two restrictions: downward programs, D-Datalog$(\downarrow, \downarrow^+)$, and child-only programs, Datalog$(\downarrow)$, obtaining an almost complete picture (see Table I, page 4). Most importantly, we showed that over unranked trees containment is decidable for downward programs (2-EXPTIME-complete and EXPSPACE-complete in the linear case) and linear child-only programs (in 3-EXPTIME and 2-EXPTIME-hard), but is undecidable for non-linear child-only programs. As we have mentioned, the 3-EXPTIME upper bound can be actually lowered to 2-EXPTIME, using techniques based on clique-width [Bojańczyk et al. 2015].

We also investigated connections between monadic datalog and extensions of regular tree pattern queries, discovering natural translations between the two formalisms; translations to RTPQs involve exponential blow-up, but translations to datalog are polynomial. Thus, results on datalog give direct corollaries about corresponding classes of RTPQs. In particular, the result on linear D-Datalog$(\downarrow, \downarrow^+)$ gives an EXPSPACE upper bound for containment of RTPQ$(\downarrow, \downarrow^+)$ queries, which solves an open problem from [Abiteboul et al. 2011].

Like all our lower bounds, the ones for Datalog$(\downarrow)$ work for containment in UCQs, but only as long as we work with Boolean queries: they do not carry over to unary queries. For example, over unranked trees, containment of unary linear Datalog$(\downarrow)$ programs in unary UCQ$(\downarrow)$ queries has been recently shown to be EXPSPACE-complete [Mazowiecki et al. 2015], while for Boolean queries we have 2-EXPTIME lower bound.

There is also room for improvement in the intersection of the two fragments: for linear D-Datalog$(\downarrow)$ we have only PSPACE-hardness and EXPSPACE upper bound.

We have worked exclusively with connected programs. Is this restriction necessary? Our algorithm for linear Datalog$(\downarrow)$ over unranked trees can be extended to cover non-connected programs, but it further complicates the already involved argument; the same extension is effortless in the alternative approach of [Bojańczyk et al. 2015]. For the remaining algorithms this question is open.

We have also ignored the sibling order. When sibling order is available in the form of the next-sibling and the following-sibling predicate, we can encode words horizontally as children of the root, which gives undecidability even for linear programs that do not use child nor descendant. If we allow only one of these predicates, we can use the negative program to say that each tree node has at most k children. Hence, undecidability carries over from ranked trees and words to unranked trees, leaving only the case of linear programs without descendant, which might be decidable. Also, the decidability results for child-only fragment on ranked trees could be extended to programs using additionally next-sibling because equality tests would remain local. Another direction to follow is datalog with non-strict order (the following-sibling-or-self predicate), which cannot limit the number of children.

ACKNOWLEDGMENTS

First and foremost we thank Pierre Bourhis for inspiring discussions and for encouraging us to generalize our results to non-linear programs, and Anca Muscholl for insights that allowed us to tighten the upper bounds for the ranked case. We also thank the anonymous referees for their detailed and helpful comments. We acknowledge the support of grant 2013/09/N/ST6/01170 from Poland's National Science Center.

REFERENCES

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

Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison Wesley.

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

François Bancilhon and Raghu Ramakrishnan. 1986. An Amateur's Introduction to Recursive Query Processing Strategies. In *Proc. SIGMOD 1986*. ACM Press, 16–52.

Michael Benedikt, Pierre Bourhis, and Pierre Senellart. 2012. Monadic Datalog Containment. In *Proc. ICALP 2012*. 79–91.

Michael Benedikt, Wenfei Fan, and Floris Geerts. 2008. XPath satisfiability in the presence of DTDs. *J. ACM* 55, 2 (2008).

Mikolaj Bojańczyk, Filip Murlak, and Adam Witkowski. 2015. Containment of Monadic Datalog Programs via Bounded Clique-Width. In *Proc. ICALP 2015*. 427–439.

Piero A. Bonatti. 2004. On the Decidability of Containment of Recursive Datalog Queries – Preliminary report. In *Proc. PODS 2004*. 297–306.

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

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

Surajit Chaudhuri and Moshe Y. Vardi. 1992. On the Equivalence of Recursive and Nonrecursive Datalog Programs. In *Proc. PODS 1992*. 55–66.

Stavros S. Cosmadakis, Haim Gaifman, Paris C. Kanellakis, and Moshe Y. Vardi. 1988. Decidable Optimization Problems for Database Logic Programs (Preliminary Report). In *Proc. STOC 1988*. 477–490.

Bruno Courcelle and Stephan Olariu. 2000. Upper bounds to the clique width of graphs. *Discrete Applied Mathematics* 101, 1-3 (2000), 77–114.

Diego Figueira. 2009. Satisfiability of downward XPath with data equality tests. In *Proc. PODS 2009*. 197–206.

André Frochaux, Martin Grohe, and Nicole Schweikardt. 2014. Monadic Datalog Containment on Trees. In *Proc. AMW 2014*.

Haim Gaifman, Harry G. Mairson, Yehoshua Sagiv, and Moshe Y. Vardi. 1993. Undecidable Optimization Problems for Database Logic Programs. *J. ACM* 40, 3 (1993), 683–713.

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

Georg Gottlob, Christoph Koch, and Klaus U. Schulz. 2006. Conjunctive queries over trees. *J. ACM* 53, 2 (2006), 238–272.

Leonid Libkin. 2003. Expressive power of SQL. *Theor. Comput. Sci.* 296, 3 (2003), 379–404.

Filip Mazowiecki, Filip Murlak, and Adam Witkowski. 2014. Monadic Datalog and Regular Tree Pattern Queries. In *Proc. MFCS 2014*. 426–437.

Filip Mazowiecki, Joanna Ochremiak, and Adam Witkowski. 2015. Eliminating Recursion from Monadic Datalog Programs on Trees. In *Proc. MFCS 2015*.

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

Jeffrey F. Naughton. 1989. Data independent recursion in deductive databases. *J. Comput. System Sci.* 38, 2 (1989), 259–289.

Frank Neven and Thomas Schwentick. 2006. On the complexity of XPath containment in the presence of disjunction, DTDs, and variables. *Log. Meth. Comput. Sci.* 2, 3 (2006).

Oded Shmueli. 1993. Equivalence of Datalog Queries is Undecidable. *J. Log. Program.* 15, 3 (1993), 231–241.

Moshe Y. Vardi. 1998. Reasoning about The Past with Two-Way Automata. In *Proc. ICALP 1998*. 628–641.