Monadic datalog and regular tree pattern queries

Filip Mazowiecki, Filip Murlak, and Adam Witkowski

University of Warsaw

Abstract. 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 descendant relation is allowed or not: 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 decidable only for linear programs. With descendant it becomes decidable over unranked trees under restriction to downward programs: in each rule we only go down from the node in the head. This restriction is motivated by the formalism of regular tree pattern queries, recently proposed in the area of ActiveXML, which we show to be equivalent to linear downward programs.

1 Introduction

Among logics with fixpoint capabilities, one of the most prominent is datalog, obtained by adding fixpoint operator to unions of conjunctive queries (positive existential first order formulae). Datalog originated as a declarative programming language, but later found many applications in databases as a query language. Unfortunately, with increased expressive power comes undecidability of basic properties of queries. Classical static analysis problems of containment and equivalence are undecidable [18]. It is also undecidable if a given datalog program is equivalent to some non-recursive datalog program [13] (equivalence to a given non-recursive program is decidable [10]). Since these problems are subtasks in important data management tasks like query minimization, data integration, or data exchange, the negative results for full datalog fuel interest in restrictions [5,7,8]. Important restrictions include *monadic* programs, using only unary intensional predicates; *linear* programs, with at most one use of an intensional predicate per rule; and *connected* programs, where within each rule all mentioned nodes are connected with each other.

When the class of considered structures is restricted to words or trees, even satisfiability is undecidable (see e.g. [1]), but for monadic datalog programs we regain decidability of containment [14]. This decidability result does not carry over to the case when tree nodes carry labels from an infinite alphabet that can be tested for equality. In this setting it matters whether descendant relation is allowed or not: descendant relation can be easily eliminated from monadic programs only if they do not use label equality. In a recent paper by Abiteboul *et al.* [1], it is shown that with descendant 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.

In this paper we show that on trees over infinite alphabet

1. containment is undecidable for child-only monadic programs, but it becomes decidable under restriction to linear programs;
2. with descendant it becomes decidable under restriction to *downward* monadic programs: in each rule we only go down from the node in the head.

Complexities are high, 3-EXPTIME for child-only programs and 2-EXPTIME for downward programs, but can be lowered under additional assumptions. To provide a broader background to these results, we first consider ranked trees, where the number of children is fixed. There, like for words, containment of monadic programs is decidable without descendant, but with descendant it is undecidable even for downward linear programs.

While forbidding descendant is a natural restriction (pointed out as an open problem in [1]), downward programs may seem exotic. But in fact, they were our initial point of interest. Our original motivation was a formalism of regular tree pattern queries (essentially, regular expressions over the set of tree patterns using child and descendant axes and data equalities), recently proposed in the area of ActiveXML [3]. As we show, this formalism is equivalent to linear downward monadic programs (a special case of a more general correspondence between monadic programs and a natural extension of RTPQs). Our decidability result solves an open problem related to ActiveXML systems.

In [3], the authors reduce equivalence testing of ActiveXML systems to containment of (unary) RTPQs over documents satisfying a Boolean combination of (Boolean) RTPQs. Then they show that this problem is in EXPTIME for RTPQs with restricted data comparisons, dubbed XPath-joins, which mimic data comparisons allowed in XPath, an XML query language used widely in practice and extensively studied (see e.g., [6,12,16,17]). Their result essentially relies on Figueira's EXPTIME-completeness of satisfiability for RegXPath($\downarrow$, =), i.e., XPath with child axis and data equality, extended with Kleene star [12]. Our result on downward datalog can be seen as an extension of Figueira's result, immediately giving decidability of the general ActiveXML problem.

Organization In Section 2 we introduce datalog and RTPQs, and explore the connections between them. Then, after a brief glance at containment over ranked trees in Section 3, we move on to unranked trees to discuss our main results in Section 4. We conclude in Section 5 with possible directions for future research. All the missing proofs can be found in the appendix, available on-line.

2 Datalog and RTPQs

In this section we introduce the two query formalisms and establish the connection between them. For datalog we only briefly recall the syntax and give informal description of the semantics; for formal semantics and background we refer the reader to [2] or [9].

Both query formalisms work over finite unranked trees labeled with letters from an infinite alphabet Σ. For a tree t, we write $nodes_t$ for the set of nodes of t and

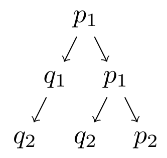
$lab_t : nodes_t \to \Sigma$ for the function assigning to each node its label. We use standard notation for axes: $\downarrow, \downarrow_+, \uparrow, \uparrow^+$ stand for child, descendant, parent and ancestor relations. Binary relation $\sim$ holds between nodes with identical labels. We also have a unary predicate a for each $a \in \Sigma$, holding for the nodes labeled with a.

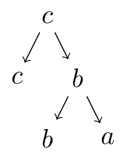
Many papers on tree-structured databases work with a slightly different data model, called "data trees", where each node has a label from a finite alphabet and a data value from an infinite data domain. In that model labels can be used explicitly in the formulas, but cannot be directly tested for equality, 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 quite 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 it does not affect decidability. For the undecidable fragments of datalog we give an additional sketch of the proof for "data trees" in the appendix.

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 one designated rule called the **goal** of the program. If there are variables in the body that are not used in the head, we assume they are quantified existentially over the nodes of the tree. 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 $\Sigma_{\mathcal{P}}$. The remaining predicates are called **intensional**. They are not stored in the database, but are defined by the program. 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.

Example 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).

$$\begin{array}{ll}
\underline{P(X)} \leftarrow X{\downarrow}Y, P(Y), X{\downarrow}Y', X \sim Y', Q(X) & (p_1) \\
P(X) \leftarrow a(X) & (p_2) \\
Q(X) \leftarrow X{\downarrow}Y, Q(Y) & (q_1) \\
Q(X) \leftarrow b(X) & (q_2)
\end{array}$$


$$\begin{tikzcd}
 & & p_1 \arrow[dl] \arrow[dr] & & \\
 & q_1 \arrow[dl] & & p_1 \arrow[dl] \arrow[dr] & \\
q_2 & & q_2 & & p_2
\end{tikzcd}$$


$$\begin{tikzcd}
 & c \arrow[dl] \arrow[dr] & & \\
c & & b \arrow[dl] \arrow[dr] & \\
 & b & & a
\end{tikzcd}$$

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

In this paper we consider only **monadic** programs, i.e., programs whose intensional predicates are at most unary. Moreover, throughout the paper we assume that the programs do not use 0-ary intensional predicates. For general programs this is merely for the uniformity of constructions and proofs: 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 is important.

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 used in the head of r. The program from Example 1 is connected and downward, but not linear. In fact, each downward program is connected.

Previous work on datalog on arbitrary structures often considered the case of connected programs [11,13]. A practical reason is that "real life" programs tend to be connected and linear (cf. [4]). Also, rules that are not connected combine pieces of unrelated data; they correspond to the algebraic operation of *cross product*, an unnatural operation in a database context. It seems even more natural to assume connectedness, as we work with tree-structured databases. We shall do so.

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 shall 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 $\exists x_1 \dots x_k \varphi$, where φ is a conjunction of atoms. Sometimes we will also speak of **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 $\mathsf{CQ}(\downarrow, \downarrow_+)$, $\mathsf{CQ}(\downarrow)$, $\mathsf{UCQ}(\downarrow, \downarrow_+)$, $\mathsf{UCQ}(\downarrow)$.

We now move to the second formalism. Let $\Delta = \{x_1, x_2, \dots\}$ be an infinite set of variables and let $\tau \subseteq \{\downarrow, \downarrow_+, \uparrow, \uparrow^+\}$ be a non-empty set of axes. 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 $vertices_\pi$ for the set of vertices of pattern π and $lab_\pi : vertices_\pi \to \Sigma \cup \Delta$ for the labeling function. Additionally, each pattern π has two distinguished nodes: in and out, denoted by $in(\pi)$ and $out(\pi)$.

Definition 1. *A* **homomorphism** $h : \pi \to t$ *from a pattern* π *over* τ *to a tree* t *is a function* $h : vertices_\pi \to nodes_t$ *such that for all vertices* v, w *in* π

– *if* $lab_\pi(v) \in \Sigma$*, then* $lab_t(h(v)) = lab_\pi(v)$*;*
– *if* $lab_\pi(v) = lab_\pi(w) \in \Delta$*, then* $h(v) \sim h(w)$*; and*
– h *preserves binary relations from* τ*.*

We write $t, v, w \models \pi$ if there is a homomorphism from π to t that maps $in(\pi)$ to v and $out(\pi)$ to w. If this is the case we also write $t, v \models \pi$ and $t \models \pi$.

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. 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 the 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 $i = 1, \dots, n$ such that $h_i(out(\pi_i)) = h_{i+1}(in(\pi_{i+1}))$ 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 $in(\pi_1)$ to v; and $t, v, w \models \Pi$ if additionally h maps $out(\pi_n)$ to w.

Definition 2. *A* **regular tree pattern query** *(RTPQ) φ over τ is a regular expression over the alphabet* $\mathsf{TPQ}(\tau)$. *By $L(\varphi)$ we denote 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. The following example shows the power of RTPQs. We skip $\downarrow$ in the expressions to ease notation. We shall simulate a counter, enumerating all values between 0 and $2^n - 1$. The RTPQ will have size $\mathcal{O}(n)$ and will use only two symbols from the alphabet. The counter value will be stored in binary in n symbols. To increase the counter correctly, we need to find the least significant position with 0. Then we can change the symbol at this position to 1 and all less significant bits to 0 (as there were only 1's on less significant positions). All positions to the left from the first 0 remain unchanged. To increase the counter when the least significant 0 is on the ith position, we use pattern

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

for $i < n$, and for $i = n$ we take $inc_n = (0)^{in} 1 \cdots 1 (1)_{out} 0 \cdots 0$. Then, we combine patterns inc_i into a counter going from 0 to $2^n - 1$:

$$c_n = val_0 \cdot \big(inc_1 \cup inc_2 \cup \cdots \cup inc_n\big)^* \cdot val_{2^n - 1}$$

where val_k is number k stored in n bits in binary, e.g., $val_0 = (0)^{in}_{out} 0 \cdots 0$. 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 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 pattern. 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 $\uparrow$.

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, translation from patterns to CQs involves quadratic blowup. As observed by Gottlob and Koch [15], each satisfiable graph pattern can be expressed as a union of TPQs, essentially reflecting possible 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 these two facts, the following result is almost immediate.

Proposition 1. *The following classes of queries have the same expressive power:*

- $\mathsf{RTPQ}(\downarrow, \downarrow_+, \uparrow, \uparrow^+)$ *and* $\mathsf{L\text{-}Datalog}(\downarrow, \downarrow_+)$*;*
- $\mathsf{RTPQ}(\downarrow, \uparrow)$ *and* $\mathsf{L\text{-}Datalog}(\downarrow)$*;*
- $\mathsf{RTPQ}(\downarrow, \downarrow_+)$ *and* $\mathsf{DL\text{-}Datalog}(\downarrow, \downarrow_+)$*;*
- $\mathsf{RTPQ}(\downarrow)$ *and* $\mathsf{DL\text{-}Datalog}(\downarrow)$.

Translations to datalog are polynomial, and translations to RTPQs are exponential.

Notice that when rules are translated into patterns, and *vice versa*, variables correspond to nodes. Moreover, 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. Therefore, 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; 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.

Recall that proof trees of linear programs are single branches. Hence, 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: simply add an ϵ-edge from the out node of each pattern to the in node of the following pattern in the word. This gives a pattern representation for proof trees of linear programs.

This representation generalizes to proof trees of arbitrary programs. For a rule r of $\mathcal{P}$ let π_r be the graph pattern corresponding to the conjunction of extensional atoms in the body of r, and let $\mathcal{G}_\mathcal{P}$ be the set of all π_r's. Then, to 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 in witnessing patters, each maximal connected ϵ-free subpattern corresponds to a rule of the program.

3 Ranked trees

Throughout the paper we work with the Boolean variant of containment, i.e., satisfiability of $\mathcal{P} \wedge \neg\mathcal{Q}$, where $\mathcal{P}$ and $\mathcal{Q}$ are treated as Boolean queries, with all variables quantified out existentially. Using well known methods one can reduce the unary variant to the Boolean one (not by simply rewriting queries, though).

To warm up, we first look at the case of words, where many ideas can be illustrated without much of the technical difficulty of trees. Over words, the relations $\downarrow$ and $\downarrow_+$ are interpreted as "next position" and "following position".

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 [1, 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 [1] 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 (see Appendix E). Once descendant is disallowed, we immediately regain decidability.

Proposition 2. *Containment over words is*
1. *undecidable for* $\mathsf{LD\text{-}Datalog}(\downarrow, \downarrow_+)$ *(even containment in UCQs);*
2. *in* EXPSPACE *for* $\mathsf{Datalog}(\downarrow)$ *and* PSPACE*-complete for* $\mathsf{L\text{-}Datalog}(\downarrow)$.

Proof. An easy reduction based on the binary counter encoding from Sect. 2 shows that even satisfiability for $\mathsf{L\text{-}Datalog}(\downarrow)$ programs is PSPACE-hard (see Appendix). Here we show that the containment for $\mathsf{L\text{-}Datalog}(\downarrow)$ programs is in PSPACE.

Let $\mathcal{P}, \mathcal{Q} \in \mathsf{L\text{-}Datalog}(\downarrow)$. Recall that $\mathcal{G}_\mathcal{P}$ is the set of patterns corresponding to the bodies of rules from $\mathcal{P}$ and for every pattern $\pi \in \mathcal{G}_\mathcal{P}$, $in(\pi)$ and $out(\pi)$ are the nodes corresponding to the head variable and the variable in the intensional atom in the rule giving rise to π. Since we work over words and we disallow $\downarrow_+$ (following position), each pattern from $\mathcal{G}_\mathcal{P}$ is equivalent to a word with the in and out nodes placed in arbitrary positions. We shall assume they are in this form. Since the programs are linear, the witnessing patterns are essentially words as well: concatenations of words from $\mathcal{G}_{\mathcal{P}}$ (with ϵ-edges connecting the out node of the preceding pattern with the in node of the subsequent pattern). We denote the set of words corresponding to witnessing patterns by $L(\mathcal{P})$. We write $L_{pref}(\mathcal{P}), L_{inf}(\mathcal{P}), L_{suf}(\mathcal{P})$ for the languages of prefixes, infixes, and suffixes of words from $L(\mathcal{P})$.

Observe that satisfiability of $\varphi = \mathcal{P} \wedge \neg\mathcal{Q}$ can be verified over finite alphabet $\Sigma_0 = \Sigma_\varphi \,\dot\cup\, \{a_1, a_2, \ldots, a_n\}$, where Σ_φ is the set of labels used explicitly by rules in $\mathcal{P}$, $\mathcal{Q}$; and n is the maximal number of nodes in a pattern in $\mathcal{G}_{\mathcal{P}} \cup \mathcal{G}_{\mathcal{Q}}$. Indeed, one can relabel any model of φ so that it uses only labels from Σ_0. Simply process the word from left to right, replacing each letter not in Σ_φ with the *least* recently used letter a_i: when an occurrence of a letter b is to be replaced by a_i, first change all later occurrences of a_i to a fresh letter, then replace all later occurrences of b with a_i. This relabeling policy ensures that equality of labels within distance n is not affected.

For program $\mathcal{P}$ we construct a deterministic finite automaton $\mathcal{A}_{\mathcal{P}}$ that recognizes words over alphabet Σ_0 satisfying $\mathcal{P}$. Let w be the whole word read by $\mathcal{A}_{\mathcal{P}}$ and let w_s be the recently read n-letters suffix. States of $\mathcal{A}_{\mathcal{P}}$ have two components. The first component is a word from $\bigcup_{k \leq n} (\Sigma_0)^k$, corresponding to the suffix w_s (initially, there are less then n letters to remember). The second component is a subset of

$$\{1, 2, \ldots, n+1\} \times \mathcal{G}_{\mathcal{P}} \times \{1, 2, \ldots, n+1\} \times \mathcal{G}_{\mathcal{P}} \,.$$

For $p, p' \leq n$, each element (p, π, p', π') of this set corresponds to a subpattern $\Pi \in L_{inf}(\mathcal{P})$, which starts from π, ends with π', and a homomorphism $h : \Pi \to w$ that maps $in(\pi)$ to $w[|w| - |w_s| + p]$ and $out(\pi')$ to $w[|w| - |w_s| + p']$. The intended meaning is that the state q stores the information about the first and last subpattern of Π from $\mathcal{G}_{\mathcal{P}}$ for every homomorphism from patterns in $L_{inf}(\mathcal{P})$ to w such that at least one node from the first and last subpattern is mapped to w_s. This information is sufficient to build information about larger patterns by induction.

Note that there is an additional value $n{+}1$ available on the first and third coordinate. In the first coordinate, value $n{+}1$ is used to indicate that the represented homomorphism is from a subpattern $\Pi \in L_{pref}(\mathcal{P})$ to w, and we only remember the information about the last part of Π. In the third component, value $n + 1$ is used to indicate that $\Pi \in L_{suf}(\mathcal{P})$, and this pattern does not have to be extended any more from the other side. If a state contains a tuple with the first and third entry equal to $n + 1$, it means that the represented Π belongs to $L(\mathcal{P})$, and so the represented homomorphism is from a witnessing pattern. Such states are accepting.

The automaton starts in state $(\varepsilon, \emptyset)$. When it reads letter a in state (u, Φ), it moves to state (v, Ψ), where v is the suffix of ua of length at most n and Ψ is defined as follows. Set Ψ contains all elements of Φ, but the first and third coordinates that are at most n are decreased by 1 (unless $|v| < n$). If either drops below 1, the element is discarded. Moreover, for all $\pi \in \mathcal{G}_{\mathcal{P}}$, if there is a homomorphism η from π to w_s and the image contains the new letter, then Ψ contains all elements $(\eta(in(\pi)), \pi, \eta(out(\pi)), \pi)$. If π corresponds to a non-recursive rule, we also add tuple $(\eta(in(\pi)), \pi, n + 1, \pi)$. If π corresponds to the goal rule, we add tuple $(n{+}1, \pi, \eta(out(\pi)), \pi)$. If both of these hold, we add tuple $(n + 1, \pi, n + 1, \pi)$. Finally, we close Ψ under legal concatenations: if Ψ

contains (p_1, π_1, p_2, π_2) and (p_3, π_3, p_4, π_4) such that $p_2 = p_3$ and $\pi_2 \cdot \pi_3 \in L_{inf}(\varphi_i)$, then Ψ should also contain (p_1, π_1, p_4, π_4).

We construct an analogous automaton for $\mathcal{Q}$. Since these automata are deterministic, we easily construct a product automaton equivalent to $\mathcal{P} \wedge \neg\mathcal{Q}$. The size of $\mathcal{A}_\mathcal{P}$ is exponential in size of $\mathcal{P}$, but states and transitions of $\mathcal{A}_\mathcal{P}$ can be generated on the fly in polynomial space. To check its emptiness we make a simple reachability test, which is in NLogSpace. Altogether, this gives a PSpace algorithm. □

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

Theorem 1. *Over ranked trees containment is*

1. *undecidable for* LD-Datalog$(\downarrow, \downarrow_+)$ *(even containment in UCQs);*
2. *in* 3-ExpTime *for* Datalog$(\downarrow)$ *and* 2-ExpTime*-complete for* L-Datalog$(\downarrow)$.

Note that from undecidability for LD-Datalog$(\downarrow, \downarrow_+)$ over words we immediately get undecidability for L-Datalog$(\downarrow, \downarrow_+)$ over trees (ranked or unranked): we simply run the programs up the tree (when we only go up, a tree looks like a word); showing undecidability for LD-Datalog$(\downarrow, \downarrow_+)$ over ranked trees requires additional effort.

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 L-Datalog$(\downarrow, \downarrow_+)$, remains undecidable as explained in Section 3, but the reduction for LD-Datalog$(\downarrow, \downarrow_+)$ in Theorem 1 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 D-Datalog$(\downarrow, \downarrow_+)$. More surprisingly, we lose decidability for Datalog$(\downarrow)$: using non-linearity and recursion we can simulate $\downarrow_+$ to a point sufficient to repeat (upwards) the reduction for words.

Proposition 3. *Over unranked trees containment of* Datalog$(\downarrow)$ *programs in* UCQ$(\downarrow)$ *queries is undecidable.*

In Section 4.2 we show how to get decidability for *linear* Datalog$(\downarrow)$ programs.

4.1 Downward programs

The aim of this section is to sketch out the proof of the following theorem.

Theorem 2. *Over unranked trees containment is 2-*ExpTime*-complete for* D-Datalog$(\downarrow)$ *and* D-Datalog$(\downarrow, \downarrow_+)$*, and* ExpSpace*-complete for* LD-Datalog$(\downarrow, \downarrow_+)$.

Notice that in the non-linear case we do not need $\downarrow_+$ to show hardness. This is not so surprising: already in Proposition 3 we simulate $\downarrow_+$ using non-linearity.

The proof of the positive claims of Theorem 2 bears some similarity to the argument showing decidability of containment of datalog programs in UCQs over arbitrary structures [10]. 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 homomorphisms (injective up to ϵ-edges) into unranked trees, and we make the labels as different from each other as permitted by $\mathcal{P}$.

Definition 3. *A tree is a* **canonical model** *for a satisfiable pattern $\pi \in \mathsf{TPQ}(\downarrow, \downarrow_+, \epsilon)$, 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 variables, connected with $\downarrow$ edges.*

Note that each 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 1. *$\mathcal{P} \wedge \neg\mathcal{Q}$ is satisfiable over unranked trees iff it is satisfiable in a canonical model of a witnessing pattern of $\mathcal{P}$.*

Let N be the maximal number of variables in a rule in $\mathcal{P}$. We write $\mathsf{TPQ}^{\mathcal{P}}(\downarrow, \downarrow_+, \epsilon)$ for the family 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. Since $\mathcal{P}$ is downward, witnessing patterns of $\mathcal{P}$ are clearly elements of $\mathsf{TPQ}^{\mathcal{P}}(\downarrow, \downarrow_+, \epsilon)$.

Patterns in $\mathsf{TPQ}^{\mathcal{P}}(\downarrow, \downarrow_+, \epsilon)$ can be viewed as trees over the alphabet $(\Sigma_{\mathcal{P}} \cup \Delta_0) \times \{\downarrow, \downarrow_+, \epsilon\}$, 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 2 is the following lemma.

Lemma 2. *The set of patterns π from $\mathsf{TPQ}^{\mathcal{P}}(\downarrow, \downarrow_+, \epsilon)$ such that $\mathcal{P} \wedge \neg\mathcal{Q}$ is satisfied in a canonical model of π is recognized by a double exponential tree automaton $\mathcal{A}_{\mathcal{P} \wedge \neg\mathcal{Q}}$; states and transitions of $\mathcal{A}_{\mathcal{P} \wedge \neg\mathcal{Q}}$ can be enumerated in exponential working memory.*

To test satisfiability of $\mathcal{P} \wedge \neg\mathcal{Q}$, we generate $\mathcal{A}_{\mathcal{P} \wedge \neg\mathcal{Q}}$ and test its emptiness. Since emptiness can be tested in PTIME, this gives 2-EXPTIME algorithm for containment.

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 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 at most N-ary trees of the special shape, emptiness of a given automaton $\mathcal{B}$ can be tested by a non-deterministic 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., [17]). In our case, the algorithm can keep guessing the main path bottom-up, together 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 $\mathcal{A}_{\mathcal{P} \wedge \neg \mathcal{Q}}$ 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 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 3. *Over unranked trees containment for* $\mathsf{L\text{-}Datalog}(\downarrow)$ *is in 3-*EXPTIME*; containment of* $\mathsf{L\text{-}Datalog}(\downarrow)$ *programs in* $\mathsf{UCQ}(\downarrow)$ *queries is 2-*EXPTIME*-complete.*

The general strategy of the proof is as for downward programs, but this time we face a new difficulty: witnessing patterns are not trees any more and they need not admit injective homomorphisms into trees. Indeed, rules of $\mathsf{L\text{-}Datalog}(\downarrow)$ programs can be turned into tree patterns by merging nodes sharing a child, and witnessing patterns are sequences of tree patterns connected with ϵ-edges between out nodes and in nodes—but this time in nodes need not be roots.

The first step to fix this is to adjust canonical models: in item 3 of Definition 3 we merge not only nodes connected with ϵ-edges, but also nodes with a common child; item 5 becomes void. After this modification we can reprove Lemma 1 for $\mathsf{Datalog}(\downarrow)$ programs (linearity plays no role here).

Unlike for downward programs, each witnessing pattern has a unique canonical model (up to the choice of fresh labels). Hence, to test satisfiability of $\mathcal{P} \wedge \neg \mathcal{Q}$ we need to find a witnessing pattern for $\mathcal{P}$ whose canonical model does not satisfy $\mathcal{Q}$. But this is also more involved than for downward programs: nodes arbitrarily far apart in the witnessing pattern may represent the same node of the canonical model. An automaton cannot compute this correspondence; we need to make it explicit.

One can think of an $\mathsf{L\text{-}Datalog}(\downarrow)$ program as a tree-walking automaton that moves 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 the **spine** of the current rule: the shortest path between the in and out nodes of the corresponding tree pattern. Let $Spine(\mathcal{P})$ be the disjoint union of the nodes in the spines of $\mathcal{P}$'s rules. In our encoding of witnessing patterns (Figure 1), an unranked tree over the alphabet $\big(Spine(\mathcal{P}) \cup \{\top\}\big)^2$ represents the way the sequence of spines is mapped to the corresponding canonical model (the remaining nodes can be computed on the fly by the automaton); $\top$ is used in nodes traversed once, only going down or only going up. The proof of Theorem 3 amounts to showing the following lemma.

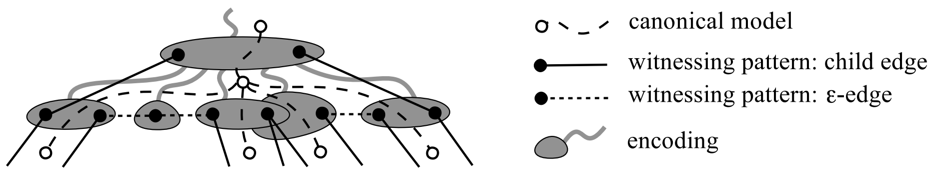


Fig. 1: Encoding of witnessing patterns

Lemma 3. *The set of encodings 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* $\mathcal{Q} \in \mathsf{UCQ}(\downarrow)$.

5 Conclusions

The containment problem for connected monadic datalog on trees over infinite alphabet is undecidable. We considered two restrictions: downward programs, $\mathsf{D\text{-}Datalog}(\downarrow, \downarrow_+)$, and child-only programs, $\mathsf{Datalog}(\downarrow)$. Table 1 summarizes our results. It is not difficult to extend the algorithm for linear $\mathsf{Datalog}(\downarrow)$ over unranked trees to cover non-connected programs; for the remaining algorithms this remains an open question.

	Unranked trees		Ranked trees	
	linear	non-linear	linear	non-linear
$\mathsf{D\text{-}Datalog}(\downarrow, \downarrow_+)$	EXPSPACE	2-EXPTIME	UNDEC.	UNDEC.
$\mathsf{Datalog}(\downarrow)$	in 3-EXPTIME	UNDEC.	2-EXPTIME	in 3-EXPTIME

Table 1: Complexity of containment for datalog fragments.

Our results yield matching upper bounds for the satisfiability of Boolean combinations of programs: using disjunctive normal form we reduce this problem to satisfiability of conjunctions of programs and negated programs, and by De Morgan laws and closure under disjunction we simplify it to conjunctions of multiple programs and a single negated program. For non-linear programs all considered classes are closed under conjunctions. For linear programs they are not; but for downward programs we have the extension to nested linear programs, and for $\mathsf{Datalog}(\downarrow)$ we can directly generalize the algorithm to conjunctions of programs. Thus, the upper bounds carry over. Using standard methods we can extend them further to combinations of unary queries; in particular, to containment of unary queries modulo a combination of Boolean queries.

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. Thus, results on datalog give direct corollaries about corresponding classes of RTPQs. In particular, the result on linear $\mathsf{D\text{-}Datalog}(\downarrow, \downarrow_+)$ gives an EXPSPACE upper bound for containment of $\mathsf{RTPQ}(\downarrow, \downarrow_+)$, which solves an open problem from [3].

For child-only queries we provide few tight complexity bounds. Some gaps stand out in Table 1, but we would like to point out one that is less apparent, but more intriguing. Like all our lower bounds, the ones for $\mathsf{Datalog}(\downarrow)$ work for containment in UCQs, but these bounds do not carry over to unary queries. For example, containment over unranked trees of linear $\mathsf{Datalog}(\downarrow)$ programs in $\mathsf{UCQ}(\downarrow)$ is 2-EXPTIME-complete, but containment of unary linear $\mathsf{Datalog}(\downarrow)$ programs in unary $\mathsf{UCQ}(\downarrow)$ queries could have much lower complexity. Also, for linear $\mathsf{D\text{-}Datalog}(\downarrow)$ we have only PSPACE-hardness.

As we have mentioned, we can incorporate finite alphabet to our setting without affecting the bounds. For ranked trees we can also easily add schema information. For unranked trees schemas can bound the branching, which immediately leads to undecidability—unless we forbid descendant. Then, we might still get decidability.

Acknowledgments. First and foremost we would like to thank Pierre Bourhis for inspiring discussions and for encouraging us to generalize our results to non-linear programs. We also thank the anonymous referees for their helpful comments. We acknowledge the support of the grant DEC-2013/09/N/ST6/01170 from Poland's National Science Center.

References

1. S. Abiteboul, P. Bourhis, A. Muscholl, Z. Wu. Recursive queries on trees and data trees. ICDT 2013: 93–104.
2. S. Abiteboul, R. Hull, V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.
3. S. Abiteboul, B. ten Cate, Y. Katsis. On the equivalence of distributed systems with queries and communication. ICDT 2011: 126–137.
4. F. Bancilhon, R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. ACM SIGMOD 1986: 16–52.
5. M. Benedikt, P. Bourhis, P. Senellart. Monadic datalog containment. ICALP 2012(2): 79–91.
6. M. Benedikt, W. Fan, F. Geerts. XPath satisfiability in the presence of DTDs. *J. ACM*, 55(2), 2008.
7. P. Bonatti. On the decidability of containment of recursive datalog queries - preliminary report. PODS 2004: 297–306.
8. D. Calvanese, G. De Giacomo, M. Vardi. Decidable containment of recursive queries. *Theor. Comput. Sci.*, 336(1):33–56, 2005.
9. S. Ceri, G. Gottlob, L. Tanca. *Logic programming and databases*. Springer, 1990.
10. S. Chaudhuri, M. Vardi. On the equivalence of recursive and nonrecursive datalog programs. PODS 1992: 55–66.
11. S. Cosmadakis, H. Gaifman, P. Kanellakis, M. Vardi. Decidable optimization problems for database logic programs (preliminary report). STOC 1988: 477–490.
12. D. Figueira. Satisfiability of downward XPath with data equality tests. PODS 2009: 197–206.
13. H. Gaifman, H. Mairson, Y. Sagiv, M. Vardi. Undecidable optimization problems for database logic programs. *J. ACM*, 40(3):683–713, 1993.
14. G. Gottlob, C. Koch. Monadic datalog and the expressive power of languages for web information extraction. *J. ACM*, 51(1):74–113, 2004.
15. G. Gottlob, C. Koch, K. Schulz. Conjunctive queries over trees. *J. ACM*, 53(2):238–272, 2006.
16. G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *J. ACM*, 51(1):2–45, 2004.
17. F. Neven, T. Schwentick. On the complexity of XPath containment in the presence of disjunction, DTDs, and variables. *Logical Methods in Computer Science*, 2(3), 2006.
18. O. Shmueli. Equivalence of datalog queries is undecidable. *J. Log. Program.*, 15(3):231–241, 1993.

A Automata

We start from a brief introduction to automata. Even though we deal with unranked trees over infinite alphabet, we shall need automata on ranked trees over finite alphabet.

A (bottom-up) **tree automaton** $\mathcal{A} = \langle \Gamma, Q, \delta, F \rangle$ on at most n-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 $\delta \subseteq \bigcup_{i=0}^{n} Q \times \Gamma \times Q^i$. 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 $k \leq n$, 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.

To work on unranked trees we encode every unranked tree t into a binary tree t_b in standard fashion. First we put an order on children for every node in t. We copy the root from t to t_b and then proceed as follows. Let v be a node from t and let v_b be its copy in t_b. The two children of v_b correspond to "previous sibling" and "last child" from t. Thus we can use the same automata for the encoded unranked trees.

B Equiexpressiveness

Proof (of Proposition 1). We shall 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. We assumed 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 $\pi = (x)^{in}{\downarrow}a{\downarrow_+}(x)_{out}$ gives $G(X) \leftarrow X{\downarrow}Z, Z{\downarrow_+}Y, a(Z), X \sim Y$. Note how the repeated variable x in π is reflected by the atomic formula $X \sim Y$.

In general, let $\mathcal{A}_\varphi$ be a finite automaton over the alphabet consisting of TPQs occurring in φ, recognizing $L(\varphi)$. 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 $\mathcal{A}_\varphi$.

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. As we have mentioned, this graph pattern (seen as a binary query) can be rewritten as union of exponentially many TPQs. Let $\mathcal{T}$ be the set of all TPQs obtained by transforming $\mathcal{P}$'s bodies.

We now construct a finite automaton reading words over $\mathcal{T}$. The states of $\mathcal{A}$ are $\top, \bot$ and the intensional predicates of $\mathcal{P}$. 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 only accepting state is $\top$. The initial state is the goal of $\mathcal{P}$. We can now construct regular expression $\varphi_{\mathcal{P}}$ (of exponential size) defining the language recognized by $\mathcal{A}$. This expression is an RTPQ and (seen as a unary query) it is equivalent to $\mathcal{P}$.

Since exponential blow-up is unavoidable when translating automata to regular expressions, it is also unavoidable in translation from datalog to RTPQs. $\square$

C Ranked trees (upper bounds)

C.1 Words

In this section we present a more detailed proof of bounding the number of needed labels for Datalog($\downarrow$) programs on words. The upper bound for non-linear programs on words is in the next section as Corollary 1.

Lemma 4. *Consider conjunction $\varphi = \mathcal{P} \wedge \neg\mathcal{Q}$ where $\mathcal{P}, \mathcal{Q}$ are* Datalog($\downarrow$) *programs on words and let Σ_0 be a finite alphabet such that*

- *$\Sigma_0 \supseteq \Sigma_\varphi$, where Σ_φ is the set of letters used in $\mathcal{P}, \mathcal{Q}$;*
- *$|\Sigma_0 \setminus \Sigma_\varphi| = n$, where n is the size of the longest path G_r for all rules r in the programs $\mathcal{P}, \mathcal{Q}$.*

Then, φ is satisfiable over Σ iff φ is satisfiable 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 modify the word v into a word over the alphabet Σ_0. 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 $|l|$. Let i be the position of the next letter to process. If the letter v_i is from the alphabet Σ_φ 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 $v_i \notin \Sigma_\varphi$ and $v_i \notin l$. If $|l| < n$, we add v_i at the beginning of l and proceed to the next letter. If $|l| = n$, we modify the word v as follows: (1) let a be the last letter in l, and let b be a fresh letter that does not occur in v, (2) for all $j > i$, if $v_j = a$, we change v_j to b, (3) for all $j \geq i$, if $v_j = v_i$, we change v_j to a, and (4) we move the letter a to the beginning of the list. After processing the last letter of v we obtain a word which uses at most n letters from $\Sigma \setminus \Sigma_\varphi$. Hence, we can turn it into a word over Σ_0, without changing the equalities among positions of the word and letters from Σ_φ. Let us denote this word by u.

It is clear that the whole procedure never changes equality of labels within distance n, and does not affect labels from Σ_φ. Since $\mathcal{P}$, $\mathcal{Q}$ use only short axes, it follows that $u \models \varphi$. $\square$

C.2 Trees

This section is devoted to prove the upper bounds in Theorem 1. We denote by R the maximal arity of nodes in the trees. Throughout the proof we will assume that $\varphi = \mathcal{P} \wedge \neg \mathcal{Q}$ and we have to decide whether φ is satisfiable. To prove Theorem 1, we shall construct a bottom-up finite tree automaton that simultaneously evaluates both programs used in φ. The automaton will work on trees with symbols from a bounded alphabet Σ_0. Consequently, it will recognize only a subset of all trees satisfying φ, but we shall prove that this suffices to check whether φ is satisfiable.

The first step of our proof is bounding the number of labels required to satisfy φ.

Lemma 5. *Let R be the maximal arity of nodes in the trees. Let Σ_0 be a finite alphabet such that*

- *$\Sigma_0 \supseteq \Sigma_\varphi$, where Σ_φ is the set of letters used in φ,*
- *$|\Sigma_0 \setminus \Sigma_\varphi| = R^{|\varphi|}$.*

Then, φ is satisfiable over Σ iff φ is satisfiable over Σ_0.

Proof. This proof is similar to the proof of Lemma 4. Let n be a size of the largest rule in the programs used in φ. Let $B \subseteq \Sigma \setminus \Sigma_\varphi$ be a set of size $R^{|\varphi|}$. We will describe a procedure that traverses the tree in a top-down fashion, level by level, and changes the labels to elements of B. Note that this way the set of processed nodes 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, 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 $a \in B \cup \Sigma_\varphi$, we can finish processing v. Assume $a \notin B \cup \Sigma_\varphi$. Pick label $b \in B$ that does not appear in processed descendants of u, nor in u itself. We can always find such label b because the number of processed descendants of u (including u itself) is bounded by $\sum_{i=0}^{n-1} R^i = \frac{R^n - 1}{R-1} < R^n \leq R^{|\varphi|}$, and so is the number of labels from B used in these nodes. Let $c \in \Sigma \setminus (B \cup \Sigma_\varphi)$ be a fresh label. We now replace all appearances of b with c, but only in unprocessed descendants of node u. Observe that these nodes are separated from nodes that keep their label b by distance at least n. Next, we replace all appearances of a with b, but only in the unprocessed descendants of u. Again, the distance from these nodes to other nodes with label a or b is at least n. Thus, the modification does not affect the outcome of any label comparison done by rules of programs in φ (because they use only short axes and are connected). After all nodes are processed, all labels in t are from $B \cup \Sigma_\varphi$ and t still satisfies φ. $\square$

We are now ready to construct an automaton $\mathcal{A}_\mathcal{P}$ that evaluates a single $\mathsf{L\text{-}Datalog}(\downarrow)$ program $\mathcal{P}$ in a tree. The automaton $\mathcal{A}_\mathcal{P}$ is a finite-branching bottom-up automaton, using alphabet Σ_0 given by Lemma 5. This construction is similar to the one for datalog programs on words in Proposition 2.

First, assume that $\mathcal{P}$ is *linear*. Since $\mathcal{P}$ is linear, its proof tree is a word. Let $\bar{r} = r_1 r_2 \cdots r_m$ be a proof word witnessing that $t \models \mathcal{P}$. We denote by $\Pi_{\bar{r}}$ the witnessing pattern obtained from it; by h the homomorphism from $\Pi_{\bar{r}}$ to t; and the patterns for r_i (from which $\Pi_{\bar{r}}$ is built) by $\pi_{r_i} \in \mathcal{G}_{\mathcal{P}}$.

Take an infix of some proof word $r_{i,j} = r_i r_{i+1} \ldots r_j$ that is already matched to t. Like in the case of words we will remember only homomorphisms for the patterns π_{r_i} and π_{r_j}. Formally, the set of states $Q^{\mathcal{A}_{\mathcal{P}}}$ has two components, like in the word case. The first component is a Σ_0-labeled tree from $\mathcal{T}$, where $\mathcal{T}$ is the set of all trees with depth bounded by n and node degrees bounded by $|\mathcal{P}|$. The second component is a subset of

$$\mathcal{G}_{\mathcal{P}} \times \bar{t} \times \mathcal{G}_{\mathcal{P}} \times \bar{t},$$

where $\bar{t} = t \cup \{\top\}$ and t is the set of nodes of the tree in the first component. The additional symbol $\top$ is explained below.

Each element (p, π, p', π') represents the first and last patterns from an infix word $r_{i,j}$. Similarly as in the case of words the automaton will not know where exactly all nodes were matched. It remembers only where the in node of π and out node of π' were matched. This is respectively the nodes p and p'. Aside from infixes of proof words, we also need to store suffixes and prefixes. This is handled by the additional symbol $\top$. The states are accepting when $p = p' = \top$. The starting state is $(\epsilon, \emptyset)$.

The transition function δ works as in the word case. The main difference is updating the states. The automaton in a node v has to build a new tree t based on the trees that were in v's children (forgetting the nodes in distance bigger than n from the root v) and then update the positions of the in and out nodes.

The set $\mathcal{T}$ is bounded double exponentially and contains trees of exponential size. Thus the size of the first component of $Q^{\mathcal{A}_{\mathcal{P}}}$ is bounded double exponentially. The second component are subsets of an exponential set which is also bounded double exponentially. Therefore, $\mathcal{A}_{\mathcal{P}}$ is doubly exponential. Because $\mathcal{A}_{\mathcal{P}}$ is deterministic, the automaton for $\neg\mathcal{Q}$, $\overline{\mathcal{A}_{\mathcal{Q}}}$, can be constructed exactly the same way. The automaton $\mathcal{A}_{\varphi}$ for φ is simply $\mathcal{A}_{\mathcal{P}} \times \overline{\mathcal{A}_{\mathcal{Q}}}$. $\mathcal{A}_{\varphi}$ accepts tree t if t it is accepted by the automaton for positive program $\mathcal{P}$ and not by $\mathcal{Q}$. Automaton $\mathcal{A}_{\varphi}$ has doubly exponential number of states; its emptiness can be checked in time linear in $|\mathcal{A}_{\varphi}|$, so the whole algorithm works in 2-EXPTIME.

Let us now look at the *non-linear case*. The Lemma 5 covers the non-linear case as well. We shall modify the construction of the automaton for linear programs. There, the automaton remembered infixes of patterns associated to proof words and we achieved this by storing the first and last patterns from $\mathcal{G}_{\mathcal{P}}$ of each matching infix. In the case of non-linear programs, a witness is a proof tree (with an appropriate homomorphism from its witnessing pattern). In this case, we need to remember a subtree of a witnessing homomorphism but similarly to the linear case, it suffices to store the root of such subtree and all leaves from the branches that were not satisfied in the current subtree of t. The number of such leaves, of course cannot be limited by the size of φ. Fortunately we need only to remember which patterns from $\mathcal{G}_{\mathcal{P}}$ are needed and where are the in or out nodes mapped. Unfortunately, this requirement incurs another exponential blow-up to the size of the automaton. The first component of states is the same as in the linear case. The second component of states is a subset of:

$$\mathcal{G}_{\mathcal{P}} \times \bar{t} \times \mathbb{P}(\mathcal{G}_{\mathcal{P}} \times \bar{t}),$$

where $\mathbb{P}$ is the powerset operation. The automaton $\mathcal{A}_\varphi$ for φ is a product automaton of automata for the positive and the negated program from φ, analogously to linear case. The number of states of $\mathcal{A}_\varphi$ is triply exponential in $|\varphi|$, so the whole algorithm works in 3-EXPTIME.

Notice that using the same technique we can prove an upper bound for non-linear programs on words. Like on words it results in an exponential blow up.

Corollary 1. *The containment problem for* $\mathsf{Datalog}(\downarrow)$ *on words is in* EXPSPACE.

D Unranked trees (upper bounds)

D.1 Queries with descendant

Proof (of Lemma 1). Let us begin with a general observation ($\diamondsuit$). For a node v of a tree t, let $t.v$ be the subtree of t rooted at v. Let t' be a copy of t with $t.v$ duplicated, i.e., with an additional copy of $t.v$ also rooted at v. Then for every datalog program $\mathcal{P}$ we have $t \models \mathcal{P} \iff t' \models \mathcal{P}$.

Coming back to the proof of the lemma, let t be a model for $\mathcal{P} \wedge \neg\mathcal{Q}$ and let π be a witnessing pattern for $\mathcal{P}$ such that there is a homomorphism h from π to t. Notice that we cannot expect h to be injective since by definition we map nodes connected with ϵ to the same node. Let π' be the pattern obtained by merging and unifying nodes connected with ϵ. We say that h is ϵ-injective if it is an injective homomorphism from π'. By the observation ($\diamondsuit$), we can add new nodes to t so that h becomes ϵ-injective but $\mathcal{Q}$ remains not satisfied in t. Notice that we can remove from t all nodes that are not in the image of h and have no descendant in the image of h (this is important because of $\downarrow_+$-edges in π). The obtained tree t' is still a model for $\mathcal{P} \wedge \neg\mathcal{Q}$.

It remains to relabel the nodes. Let t_π 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'. Notice that there is a natural correspondence between the nodes in t_π and t'. Since there is a homomorphism from π to t', whenever two nodes in t_π have equal labels, the corresponding nodes in t' also have equal labels. Also, whenever a node in t_π has a label from $\Sigma_{\mathcal{P}}$ (i.e., used explicitly in $\mathcal{P}$), the corresponding node t' has the same label. This proves that $t_\pi \models \mathcal{P} \wedge \neg\mathcal{Q}$. $\square$

Proof (of Lemma 2). We first show that we can recognize the set of certain encodings of canonical models of patterns from $\mathsf{TPQ}^{\mathcal{P}}(\downarrow, \downarrow_+, \epsilon)$.

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 relabeling 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 canonical model point of view the intended meaning is that nodes labeled with $(\$, \downarrow)$ encode nodes with unique labels. Because of this we assume that there is no letter $(\$, \epsilon)$ because merging nodes could violate the uniqueness of a label. 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 $\downarrow_+$-edges 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 downward program $\mathcal{R}$ on encodings. The states of automaton $\mathcal{A}_{\mathcal{R}}$ are subsets of

$$Part(\mathcal{R}) = \bigcup_{\rho \in \mathcal{G}_{\mathcal{R}}} Part(\rho)$$

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

Suppose that the currently processed node v has label $(a, \downarrow)$. 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 ρ in fact get different labels; this gives incompatibility if they are labeled with the same variable in ρ.

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 ρ to v, whenever possible. How do we do this? Assume p is an unmatched node in ρ. We can extend f to p if and only if

– each descendant of p in pattern ρ is matched; and
– either p has label a in ρ, or p has label $x \in \varDelta$ in ρ and each *matched* node position p' with the same label x in ρ is mapped to letter 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 the children of an unmatched node p in ρ is a node position that has been matched before, then we must extend f to p. If we cannot, we discard f. If p is an unmatched $\downarrow_+$-edge in ρ, 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_a be the set of all possible extensions of matchings from M. Clearly, M_a correctly represents all partial homomorphisms from patterns in $\mathcal{G}_{\mathcal{R}}$ to the read subtree. But we are interested in homomorphisms of witnessing patterns, not just elements of $\mathcal{G}_{\mathcal{R}}$. 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. Suppose that pattern ρ corresponds to rule r of $\mathcal{R}$, and consider a node u in ρ that corresponds to a variable X in r such that r contains some intensional atoms with variable X. 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 (each root of the matched subtree). We can compute them from $\mathcal{R}$ and M_a in PTIME (simple reachability), which means time exponential in the size of $\mathcal{R}$.

If the label of the currently processed node v is (a, ϵ), we need to propagate up the tree partial matchings at the children of v, not at v itself. Hence, we should take set M for the state, except that we must replace each $b \in \varSigma_0 \setminus (\varSigma_{\mathcal{P}} \cup \{a\})$ with $\flat$ in the matchings, because b will not occur any more.

If $\mathcal{A}_{\mathcal{R}}$ ever finds a total matching of a pattern corresponding to the goal rule of $\mathcal{R}$, it moves to the accepting state $\top$.

Recognizing patterns whose canonical models satisfy $\mathcal{P} \wedge \neg\mathcal{Q}$. We define the standard product automaton $\mathcal{A}_{\mathcal{P}\wedge\neg\mathcal{Q}} = \mathcal{A}_{\mathcal{P}} \times \overline{\mathcal{A}_{\mathcal{Q}}}$, where $\overline{\mathcal{A}_{\mathcal{Q}}}$ denotes the complement of $\mathcal{A}_{\mathcal{Q}}$. 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 $\mathcal{P} \wedge \neg \mathcal{Q}$. □

D.2 Child-only queries

In this section we provide the full proof for Lemma 3 which remains to prove Theorem 3. Before we proceed with the proof we need to introduce some definitions.

Canonical models. We start with the definition of canonical models for L-Datalog($\downarrow$) programs. This definition is only a small adjustment of Definition 3 for downward programs.

Definition 4. *A tree is a* **canonical model** *for a satisfiable pattern $\pi \in \mathsf{TPQ}(\downarrow, \uparrow, \epsilon)$, 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 and merge all nodes that share a child;*
4. *substitute each variable with a fresh label.*

Like for downward programs it suffices to consider only canonical models.

Lemma 6. *$\mathcal{P} \wedge \neg \mathcal{Q}$ is satisfiable over unranked trees iff it is satisfiable in a canonical model of a witnessing pattern of $\mathcal{P}$.*

Proof. The proof of Lemma 1 essentially works also for Datalog($\downarrow$) programs. The only difference is that we have to modify the definition of ϵ-injective homomorphisms. Recall that in the downward case it had to be injective from the pattern π' obtained by merging and unifying nodes connected with ϵ. For Datalog($\downarrow$) programs we must also merge nodes sharing a child to obtain a tree pattern. After this modification the proof is the same (even easier because we do not have to deal with $\downarrow_+$-edges). □

The proof trees for linear Datalog are in fact proof words, because each rule has at most one out node. There is a one to one correspondence between proof words and witnessing patterns for $\mathcal{P}$. For this reason we can write about canonical models for proof words. For every proof word $\bar{r}$ we denote its corresponding witnessing pattern by $\Pi_{\bar{r}}$ and its corresponding canonical model by $t_{\bar{r}}$. Notice that only satisfiable proof words have a canonical model. With every satisfiable proof word we also associate a canonical homomorphism $h_{\bar{r}} : \Pi_{\bar{r}} \to t_{\bar{r}}$.

Rules and patterns. For a rule r we denote by π_r its pattern representation. Without a loss of generality we can assume that for all rules r their patterns π_r are satisfiable. Moreover, we can assume that all patterns π_r for rules in programs $\mathcal{P}$ and $\mathcal{Q}$ are trees. This is because each pattern that is a DAG not a tree can be transformed into a tree pattern for an equivalent rule in polynomial time by merging all nodes with a common child (this is an identical procedure as in step 3 from Definition 4).

We call a rule r stationary if it is non-recursive or the in and out node of r are one node. We call a pattern π_r stationary if its rule is stationary. We divide the nodes each non-stationary pattern π_r into three parts: the **spine** is are nodes from (ignoring edge direction) path in π_r between the in node and the out node, including in and out nodes; the **head** are the nodes from the path (once again ignoring edge directions) from the root of π_r to the father of the least common ancestor of the in and out nodes; and the **limbs** are all nodes of π_r that are neither in the spine nor in the head. Let $\bar{r} = r_1 r_2 \cdots r_m$ be a satisfiable proof word for $\mathcal{P}$. The spine of $\mathit{\Pi}_{\bar{r}}$ is the sum of all spines of π_{r_i}. We will use the notation $Spine(\pi_{r_i})$ and $Spine(\mathit{\Pi}_{\bar{r}})$. Let $t_{\bar{r}}$ be a canonical model for $\bar{r}$. A node $v \in t_{\bar{r}}$ is a **core** node if it belongs to $h_{\bar{r}}(Spine(\mathit{\Pi}_{\bar{r}}))$. The set of all core nodes is called the core of $t_{\bar{r}}$. Notice that it is a tree, because it is a connected subgraph of $t_{\bar{r}}$.

We say that a pattern π_{r_i} **enters** a core node v of $t_{\bar{r}}$ iff v is in the image of the spine of π_{r_i} and there is no r_j in $\bar{r}$ with $j < i$ such that v is in the image of the spine of π_{r_j}. We say that a pattern π_{r_i} **leaves** the node v if v is in the image of the spine of π_{r_i} and the out node of π_{r_i} is outside the subtree of $t_{\bar{r}}$ rooted at v. It follows from the definition of the canonical model that for each core node v there is at most one rule that enters it and at most one rule that leaves it.

Encoding of a proof word. Let $\bar{r} = r_1 r_2 \cdots r_m$ be a satisfiable proof word. Notice that a spine of π_{r_i} is either a path going down from the root or a path first going up and then down. This includes degenerated cases like a path going only up or a single node who is both the in and the out node. We define an order $\prec$ on the sibling nodes in the core of $t_{\bar{r}}$. Suppose we have two sibling nodes v, w such that v is in the image of $Spine(\pi_{r_i})$ and w is in the image of $Spine(\pi_{r_j})$. If $i < j$ then $v \prec w$. If $i = j$ then $v \prec w$ iff v is the ancestor of the in node and w is the ancestor of the out node.

We encode $\bar{r}$ in a modified core of its canonical model. Let t_{core} be the core of $t_{\bar{r}}$. We order the siblings according to $\prec$ relation. The next step is to relabel this tree.

Let $Spine(\mathcal{P}) = \bigcup_{r \in \mathcal{P}} Spine(\pi_r)$. We assume that $Spine(\pi_{r_1}) \cap Spine(\pi_{r_2}) = \emptyset$ for $r_1 \neq r_2$. Our automaton will work over trees labeled with tuples from

$$(Spine(\mathcal{P}) \cup \{\top\}) \times (Spine(\mathcal{P}) \cup \{\top\}).$$

The first component is a node mapped to v's parent from the pattern π_{r_i} that enters v. The second component is a node mapped to v's parent from the pattern π_{r_j} that leaves v. We use $\top$ if there is no appropriate position.

The idea of this encoding is to be able to capture the spine of $\mathit{\Pi}_{\bar{r}}$. To include the stationary patterns, we add new nodes to t_{core}. Let π_{r_i} be a stationary pattern. We denote by v the node to which its in node was mapped. We add to t_{core} a new child for v, labeled with the in node of π_{r_i} on both coordinates. It is easy to extend the definition of $\prec$ on newly added nodes to be consistent with the order of rules in $\bar{r}$.

The new nodes ensure that all instances of rules used in the proof word $\bar{r}$ are recoverable from the encoding. Notice that this encoding does not explicitly contain the non-core nodes of the canonical model. Of course, those nodes can be used by the negative query $\mathcal{Q}$. The automaton will be able to deduce all the necessary information about the non-core nodes from the labels of the encoding. The proposed encoding does not guarantee that the encoded proof word is satisfiable. This will be checked by the automaton.

We encoded the proof words in trees of arbitrary rank. As the last step, we encode them as binary trees using the standard method: the two children of a node are: its previous sibling and its rightmost child.

Proof (of Lemma 3). We will construct a triple exponential automaton that recognizes all canonical models for $\mathcal{P}$ that do not satisfy $\mathcal{Q}$. Unlike in Lemma 2, where we built independent automata for $\mathcal{P}$ and $\mathcal{Q}$ and took a product automaton for $\mathcal{P} \wedge \neg\mathcal{Q}$, in this proof we construct the whole automaton at once. However, the parts for $\mathcal{P}$ and $\mathcal{Q}$ are mostly independent and we describe them separately. To describe the states we need an alphabet $\Sigma_0 = \Delta_0 \cup \Sigma_{\mathcal{P}} \cup \{\bot\}$, where $\Delta_0 \subseteq \Delta$ is a set of $3n$ fresh labels; n is the size of the largest rule in $\mathcal{P}$; $\bot$ is an additional label; and $\Sigma_{\mathcal{P}}$ is the set of constants used explicitly in $\mathcal{P}$. Like in Lemma 2 the labels from Σ_0 are only representing the labels from the canonical model.

The positive fragment. For a node v, $\mathcal{A}_{\mathcal{P}\wedge\neg\mathcal{Q}}$ will calculate which part of the proof word can be realized using only the parent of v and nodes from subtrees of v's siblings preceding v (including the subtree rooted in v). This information will be stored as a pair of elements from $Spine(\mathcal{P})$, the position that enters the parent of v and the one that leaves v. Additionally, the automaton needs to remember all ancestors of v that were in the image of the head nodes of patterns used so far. There are at most n such ancestors, as patterns mapped in a node v can reach at most n nodes above it. For this we need at most n different labels to be able to distinguish labels and the constants from $\Sigma_{\mathcal{P}}$.

Let u_b be the previous sibling of v and u_c be the rightmost child of v. Let $\mathsf{enter}(u)$, $\mathsf{leave}(u)$ be the entering and leaving position assigned to u by $\mathcal{A}_{\mathcal{P}\wedge\neg\mathcal{Q}}$. Given positions $\mathsf{enter}(u_c)$, $\mathsf{leave}(u_c)$, $\mathsf{enter}(u_b)$, $\mathsf{leave}(u_b)$ and the label of v, the automaton $\mathcal{A}_{\mathcal{P}\wedge\neg\mathcal{Q}}$ checks that:

- $\mathsf{leave}(u_b)$ matches the entering position from the label of v;
- the entering position from the label of v matches $\mathsf{enter}(u_c)$;
- $\mathsf{leave}(u_c)$ matches the leaving position from the label of v;

Two positions match if the appropriate leaving and entering nodes are the same (recall that labels correspond to the nodes mapped to the parent of v) or are joined with an ϵ-edge in the corresponding witnessing pattern. Finally, $\mathcal{A}_{\mathcal{P}\wedge\neg\mathcal{Q}}$ updates the stored labels of the ancestors. It joins the requirements enforced by the stored ancestor labels from the u_b and u_c Note that this operation can merge two labels into one.

Altogether the states used for the positive query evaluation are

$$(Spine(\mathcal{P}) \cup \{\top\}) \times (Spine(\mathcal{P}) \cup \{\top\}) \times (\{\bot\} \cup \Sigma_0)^n$$

The $\top$ symbol is for nodes that do not have an entering or leaving pattern. This happens if the proof word is completed from the beginning or end and there is no need to extend it. If $\mathcal{A}_{\mathcal{P}\wedge\neg\mathcal{Q}}$ finds a node with a label different from $\top$ after reaching $\top$ on the corresponding component in the state, it rejects the tree.

Notice that until now we did not care about limb nodes, or checking if the proof word is satisfiable. All of this is described in the last part.

The negative fragment. Similarly to the automaton from the proof of Lemma 2, the building blocks for the negative query evaluation will be partial homomorphisms. Recall that

$$Part(\pi) = (\Sigma_0 \cup \{\flat, \bot\})^{Pos(\pi)} \tag{1}$$

is the set of all partial homomorphisms for a rule π, where: $Pos(\pi)$ is the set of nodes in π; $\bot$ denotes a node that is not matched yet; $\flat$ denotes a node was matched correctly, but its label will not appear in the tree anymore. In Lemma 2 if p was a node mapped to a label different from $\bot$ but its parent was mapped to $\bot$ then we assumed that p was mapped to v. Now, $\mathcal{A}_{\mathcal{P}\wedge\neg\mathcal{Q}}$ has to remember partial homomorphisms matched up to n nodes above v, so $\mathcal{A}_{\mathcal{P}\wedge\neg\mathcal{Q}}$ will store a number form 0 to n denoting how many nodes above v a partial homomorphism is matched. Let $Match(\pi) = Part(\pi) \times [0, \ldots, n]$ and $Match(\mathcal{Q}) = \bigcup_{\pi\in\mathcal{Q}} Match(\pi)$. Note that $|Match(\mathcal{Q})|$ is exponential in $|\mathcal{Q}|$.

$\mathcal{A}_{\mathcal{P}\wedge\neg\mathcal{Q}}$ will calculate all possible subwords of proof words for $\mathcal{Q}$ satisfied in the tree processed so far. The subwords of proof words for $\mathcal{Q}$ can be arbitrarily long, but we can represent each subword with a pair of partial homomorphisms like in Theorem 1. Let π_f be the pattern of the first rule and π_l the pattern of the last rule. The automaton remembers the homomorphism from π_f and the homomorphism from π_l. As $\mathcal{A}_{\mathcal{P}\wedge\neg\mathcal{Q}}$ can access the information about non-core nodes only when it finds an appropriate positive pattern, some of the patterns between π_f and π_l may not be fully mapped. Therefore, in addition to the starting and ending partial homomorphism, $\mathcal{A}_{\mathcal{P}\wedge\neg\mathcal{Q}}$ has to remember a set of partial homomorphisms that need to be found for this subword to be satisfied. The number of such objects is double exponential (in $|\mathcal{Q}|$).

The states of $\mathcal{A}_{\mathcal{P}\wedge\neg\mathcal{Q}}$ used for negative query evaluation will be:

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

Notice that if the negative query is an $\mathsf{UCQ}(\downarrow)$, then each proof word has only one rule and then we only need $Match(\mathcal{Q})$.

Given the state of the left sibling v_b and rightmost child v_c of a node v, $\mathcal{A}_{\mathcal{P}\wedge\neg\mathcal{Q}}$ can easily calculate necessary information for v.

- It changes to $\flat$ all labels not used on the path above v nor by the positive pattern that enters or leaves v. Then $\mathcal{A}$ discards those proof subwords that are no longer satisfiable;
- Then, it calculates all possible extensions of the partial homomorphisms using the states from v_b and from v_c;
- If the positive proof word was extended with a new pattern π, $\mathcal{A}_{\mathcal{P}\wedge\neg\mathcal{Q}}$ finds all matchings from $\mathcal{Q}$ to the nodes from the encoded canonical model that π is mapped to. This includes the limb and head nodes. Those matchings are used to extend existing proof subwords or to create new ones.

If in any state $\mathcal{A}_{\mathcal{P}\wedge\neg\mathcal{Q}}$ finds a complete proof word for $\mathcal{Q}$ it rejects the tree.

Using partial homomorphisms the automaton can also check whether the proof word is satisfiable. It only needs to remember how the patterns of the entering node and leaving node are labelled. While matching the leaving and entering nodes in the positive fragment it must also update the equalities between the nodes and constants. Thus the automaton needs additionally the states:

$$Match(\mathcal{P}) \times Match(\mathcal{P}).$$

To achieve this, $\mathcal{A}_{\mathcal{P}\wedge\neg\mathcal{Q}}$ has to distinguish between the labels from the entering pattern, the leaving pattern and the nodes from the path above the current node. This is where the automaton actually needs $3n$ fresh labels in $\varDelta_0$.

The full set of states of $\mathcal{A}_{\mathcal{P}\wedge\neg\mathcal{Q}}$ is:

$$\begin{aligned}(Spine(\mathcal{P}) \cup \{\top\}) \times (Spine(\mathcal{P}) \cup \{\top\}) \times (\{\bot\} \cup \varSigma_0)^n \times \\ \times Match(\mathcal{Q}) \times Match(\mathcal{Q}) \times \mathbb{P}(Match(\mathcal{Q})) \times \\ \times Match(\mathcal{P}) \times Match(\mathcal{P}).\end{aligned}$$

The accepting states of $\mathcal{A}_{\mathcal{P}\wedge\neg\mathcal{Q}}$ are those with $\top$ on both entering and leaving coordinates and $\bot^n$ (meaning no requirements) on the third coordinate, meaning no requirements for the path above the root.

The number of states of $\mathcal{A}_{\mathcal{P}\wedge\neg\mathcal{Q}}$ is triple exponential in $|\mathcal{Q}|$ so deciding satisfiability of $\mathcal{A}_{\mathcal{P}\wedge\neg\mathcal{Q}}$ gives a 3-EXPTIME algorithm for deciding if $\mathcal{P} \subseteq \mathcal{Q}$. Notice that the third exponential comes from the double power set used for the negative query evaluation. In the case of containment in UCQ($\downarrow$), the size of the automaton drops to double exponential in $|\mathcal{P}| + |\mathcal{Q}|$ and the whole algorithm works in 2-EXPTIME. □

E Lower bounds

In this section we present the hardness proofs. The reductions mostly use RTPQs; by Proposition 1 the bounds carry over to the corresponding fragments of datalog.

The PSPACE lower bound of Proposition 2, is given by Proposition 4. The 2-EXPTIME lower bound of Theorem 1 is proved in Proposition 10. The undecidability result from Proposition 2 is proved in Proposition 5. The undecidability result on from Theorem 1 is proved in Proposition 6 a corollary from Proposition 5.The EXPSPACE lower bounds of Theorem 2 are shown in Proposition 9, while 2-EXPTIME bounds follow from the lower bound on D-Datalog($\downarrow$) programs given in Proposition 10.

In our lower bounds the negative queries are non-recursive, i.e., UCQs. One should note that whenever we put restrictions on the positive query (linearity, connectedness, etc.) then the negative query also fulfils these restrictions.

We begin with words. First, we present the simple PSPACE reduction, and then the more involved undecidability result.

Let us introduce RTPQ-like notation for programs on words. We change the letter T to W: we talk about WPQs, RWPQs, WPQ(τ) etc. For such queries, instead of $\uparrow$ and $\downarrow$, we use $\rightarrow$ and $\leftarrow$, respectively; we also often skip the symbol $\rightarrow$ to enhance readability. Let us underline that the RWPQs are always considered on models restricted to words.

Proposition 4. *Satisfiability for* RWPQ($\rightarrow$) *is* PSPACE*-hard.*

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

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

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

$$(q_1)^{in} X_1 X_2 \dots \mathit{Head} \dots X_n \rightarrow \tag{2}$$
$$\rightarrow (q_2)_{out} X_1 X_2 \dots \mathit{Head}' \dots X_n \tag{3}$$

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

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

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

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

The final RWPQ will be

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

where

$$\varphi_0 = (q_0)^{in}_{out} \widehat{\perp} \perp \cdots \perp$$
$$\varphi_F = \bigcup_{q \in F} (q)^{in}_{out} X_1 X_2 \dots X_n$$

describe initial and final configurations, respectively.

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

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

Proposition 5. *Containment of* $\mathsf{RWPQ}(\rightarrow, \rightarrow^{+})$ *queries in* $\mathsf{UCQ}(\rightarrow, \rightarrow^{+})$ *is undecidable.*

Proof. To prove undecidability, we reduce from the following 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 k_0,
- a final tile k_F,

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

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

We shall construct two queries φ and ψ. Query 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 matched word is a proper encoding of some tiling.

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

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

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

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

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

The positive query φ is defined as

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

where

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

enforces the initial tile and ensures that the final tile k_F is in the last column that is present in the first row. Because of the RWPQs properties, each row can be longer than the previous one, so even if k_F is in the last row, this does not guarantee that the tiling is correct. Last row of φ_0 solves this issue.

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

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

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

ensures correctness of the all the remaining rows.

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

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

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

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

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

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

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

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

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

Other errors like this are equally easy to define. The remaining errors are the same column numbers or row numbers in different rows. To find them we need the $\to^+$ relation. We show first the queries for columns. For every $Q_1, Q_2 \in K$ we add a query:

$$\begin{aligned} \exists x_1, \ldots, x_8 \quad & x_1 \to x_2, x_2 \to x_3 \to x_4 \\ & x_4 \to^+ x_5, x_5 \to x_6, x_6 \to x_7, x_7 \to x_8 \\ & x_1 \sim x_4, x_3 \sim x_6, Q_1(x_4), Q_2(x_8) \end{aligned}$$

is that x_1, x_4 are the same row numbers and x_3, x_6 are the same column numbers. The Q_1, Q_2 predicate enforce that we compare proper labels. Now we give the query for rows.

$$\begin{aligned} \exists x_1, x_2, x_3, x_4 \quad & x_1 \to x_2, x_2 \to^+ x_3, x_3 \to x_4, \\ & \#(x_1), \#(x_3), x_2 \sim x_4 \end{aligned}$$

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

Proposition 6. *Containment of* $\mathsf{RTPQ}(\downarrow, \downarrow_+, \uparrow, \uparrow^+)$ *programs in queries from* $\mathsf{UCQ}(\downarrow, \downarrow_+)$ *is undecidable over trees.*

Proof. This is a direct consequence of Proposition 5. We change the queries used in the word case by replacing $\to$ with $\uparrow$ and $\to^+$ with $\uparrow^+$, and add $X{\downarrow_+}\bot$ at the beginning of each query. We use $\bot$ as a starting symbol for the coded word (instead of the first $\#$). The query first finds $\bot$ somewhere in the tree and then goes up. As there is only one path up in the tree, it behaves exactly as in the case of words. $\square$

In Section 4.1 we mentioned that the containment problem for downward programs becomes undecidable if the models are ranked trees. By Proposition 1 it suffices to show undecidability for containment of $\mathsf{RTPQ}(\downarrow, \downarrow_+)$ in $\mathsf{UCQ}(\downarrow, \downarrow_+)$.

Proposition 7. *Containment of* $\mathsf{RTPQ}(\downarrow, \downarrow_+)$ *programs in queries from* $\mathsf{UCQ}(\downarrow, \downarrow_+)$ *is undecidable over ranked trees.*

$$\begin{tikzcd}
 & & & C \arrow[dlll] \arrow[dl] \arrow[dr] & & \\
\flat_1 & \cdots & \flat_{R-1} & & t \arrow[dl] \arrow[dr] & \\
 & & & \flat_1 & \cdots & \flat_{R-1}
\end{tikzcd}$$

Fig. 2: $C{\downarrow}t$ modified

Proof. The queries are almost the same as in Proposition 5 where we worked on words. We slightly modify the formulas to work properly on ranked trees. We start from changing the arrows from $\rightarrow$ to $\downarrow$. Let R be the maximal rank of the trees. Our queries will use $R-1$ additional symbols $\Sigma_R = \{\flat_1, \ldots, \flat_{R-1}\}$. We modify the positive query φ from Proposition 5 by adding $R-1$ children below every node. The newly added children are labeled with symbols from Σ_R. For example the subquery $C{\downarrow}t$ is modified to query in Figure 2.

We want to enforce a unique path in our model. The negative query ψ additionally assures there are no nodes below nodes labeled with Σ_R. We add to ψ:

$$\begin{array}{ll} \exists x, y & x{\downarrow_+}y \\ & \flat_i(x) \end{array}$$

for every $i < R$. This way the query φ has to be matched to a single path (the one without nodes from Σ_R) and thus the encoding of the tiling problem is correct. $\square$

Before we show next lower bounds we introduce a helpful shorthand. The query $x{\downarrow_i}y$ is equivalent to $x{\downarrow}z_1{\downarrow}\ldots{\downarrow}z_{i-1}{\downarrow}y$ where $z_1, \ldots, z_{i-1}$ are fresh variables. In particular ${\downarrow_1} = {\downarrow}$.

We show how to modify the encoding in Proposition 5 and Proposition 6 to work for $\mathsf{Datalog}(\downarrow)$ programs.

Proposition 8. *Containment of* $\mathsf{Datalog}(\downarrow)$ *programs in queries from* $\mathsf{UCQ}(\downarrow)$ *is undecidable over unranked trees.*

Proof. We check the satisfiability of $\mathcal{P} \wedge \neg\mathcal{Q}$, where $\mathcal{P} \in \mathsf{Datalog}(\downarrow)$ and $\mathcal{Q} \in \mathsf{UCQ}(\downarrow)$. We encode the tiling problem similarly as in the Proposition 5. We shall need unique successor for each node, so the query will go up the tree. This time, the program can only compare data locally, as only the $\downarrow$ relation is allowed. To simulate a non-local label comparison, the program will copy the value needed for the comparison into a child of some node higher in the tree until it finds the other value and can perfom the comparison. Each such copying will be performed as a separate branch of the execution of the program. Note that this method works only for unranked trees as there is no limit on the number of children used for copying the labels.

Let K be the set of tiles with k_0 as the initial tile and k_F as the final tile. The relations H and V are, respectively, the horizontal and vertical relations between tiles. The alphabet is like in Proposition 5 but with an additional symbol, $\mathbb{D} = K \cup \mathbb{N} \cup \{\#, \$\}$.

Placing a tile is represented by four letters, (n, n', m, t), just like in Proposition 5. Consecutive rows are separated by $\#$'s and consecutive columns are separated by \$, i.e., the tiling is encoded in a word of the form $((n \cdot n' \cdot m \cdot t \cdot \$)^* \cdot \#)^*$. Since we work on trees this word will be one of the paths in the model tree. Like before the positive program $\mathcal{P}$ will describe constraints enforced by tiling relations, initial configuration and ending tile. The program $\mathcal{Q}$ will be used to guarantee that the matched tree is a proper encoding of some tiling.

The programs are essentially the same as queries in Proposition 5. The main difference is that we cannot use the $\uparrow^+$ relation. We show how to simulate it using non-linearity of the programs. We present it on the example of the query α from Proposition 5. The following rules for the predicate G correspond to that query.

For every $(Q_1, Q_2) \in H$ and $(Q_1, Q_3) \in V$ we add a rule:

$$\begin{aligned} G(x) \leftarrow \quad & x{\uparrow}x_2{\uparrow}x_3{\uparrow}x_4{\uparrow}x_5{\uparrow}x_6{\uparrow}x_7{\uparrow}x_8{\uparrow}x_9, \\ & \$(x_5), G(x_6), x \sim x_6, x_2 \sim x_7, Q_1(x_4), Q_2(x_9), \\ & P(x_3), \\ & y_1{\uparrow}y_2{\uparrow}y_3{\uparrow}y_4{\uparrow}y_5{\uparrow}y_6{\uparrow}y_7{\uparrow}y_8{\uparrow}y_9{\uparrow}x_5, \\ & y_1 \sim x_2, y_3 \sim x_3, Q_3(y_4), \$(y_5), U_9(y_1), \\ & y_6 \sim x_2, y_8 \sim x_8, y_7 \sim y_2 \end{aligned}$$

Figure 3 presents the above rule as a pattern. Since there is more than one intensional node we marked all intensional predicates explicitly. Like in RTPQs instead of using the $\sim$ relation we use the same variables to denote that two nodes have the same label. We changed the variables x_i, y_i into R, Q etc, so it is clear on the picture which nodes correspond to columns, rows or tiles. There are three intensional predicates used in this rule: $G(x_6), U_9(y_1), P(x_3)$. One of them – $P(x_3)$ is needed for the negative program $\mathcal{Q}$ to work and will be explained later.

The predicate U_9 is responsible for simulating the $\uparrow^+$ relation. Basically, it copies the path of 9 nodes leading to a node labeled with \$ under the next node labeled with \$.

$$\begin{aligned} U_9(x) \leftarrow \quad & x{\uparrow}x_2{\uparrow}\dots{\uparrow}x_9{\uparrow}z{\uparrow}^5 z', \\ & y{\uparrow}y_2{\uparrow}\dots{\uparrow}y_9{\uparrow}z', \\ & x \sim y, x_2 \sim y_2, \dots, x_9 \sim y_9 \$(z'), U_9(y) \end{aligned}$$

There is an additional rule for U_9 where we copy the nodes one node higher. This is because when we finish the row, there is an additional $\#$ symbol. We just put $z_1{\uparrow}z_2{\uparrow}^5 z'$ instead of $z{\uparrow}^5 z'$ and add an additional conjunct $\#(z_2)$. The only non-recursive rule for U_9 is when it finds the proper row and column number. Then we only check if all equalities are satisfied.

$$\begin{aligned} U_9(x) \leftarrow \quad & x{\uparrow}x_2{\uparrow}\dots{\uparrow}x_9{\uparrow}z{\uparrow}y{\uparrow}y_2{\uparrow}\dots{\uparrow}y_9, \\ & x \sim y, x_2 \sim y_2, \dots, x_9 \sim y_9 \end{aligned}$$

Like before we need an additional rule for U_9 with an additional node z_2 labeled $\#$ laying between z and y. To sum up, the predicate U_9 simulates $\uparrow^+$ by copying an appropriate fragment of the tree step by step until it finds another fragment where it fits.

$$\begin{tikzcd}
Q_2 & \\
C' \arrow[u] & \\
R' \arrow[u] & \\
G(R) \arrow[u] & \\
\$ \arrow[u] & \\
Q_1 \arrow[u] & Q \arrow[ul] \\
P(C) \arrow[u] & C' \arrow[u] \\
R' \arrow[u] & R'' \arrow[u] \\
(R)^{in} \arrow[u] & R' \arrow[u] \\
 & \$ \arrow[u] \\
 & Q_3 \arrow[u] \\
 & C \arrow[u] \\
 & R'' \arrow[u] \\
 & U_9(R') \arrow[u]
\end{tikzcd}$$

Fig. 3: The G rule. There are 3 intensional predicates G, P, U_9.

It is easy to rewrite rules for the other queries in Proposition 5 that used $\uparrow^+$ axis. In φ_0 we use the $\uparrow^+$ axis twice to make sure that in our encoding the rows have the same number of columns. We define a rule $U_{5,4}$ which works similar to U_9, i.e., first it copies 9 nodes up to the next \$ but instead of the non-recursive rule for U it ends matching the first 5 nodes and with a U_4 predicate for the last 4 nodes. The rules for U_4 are like for U_9 it just copies 4 nodes instead of 9 nodes.

We can use this trick in the rules of the $\mathcal{P}$ program. Unfortunately this simulation requires recursive rules thus we cannot use it in the program $\mathcal{Q}$. In Proposition 5 we needed the $\rightarrow^+$ relation to find columns and rows with the same labels. We take care of this in a different way here.

Recall the intensional predicate P used in G. We give the the rules for P now:

$$\begin{aligned} P(x) \leftarrow \quad & x{\uparrow}x_2{\uparrow}\ldots{\uparrow}x_6, \\ & \$(x_3), y{\uparrow}x_6, x \sim y, P(y); \\ P(x) \leftarrow \quad & x{\uparrow}x_2{\uparrow}\ldots{\uparrow}x_4, \\ & \#(x_4) \end{aligned}$$

The first rule ensures that every column node has all previous column labels as its children. Once the row ends the program applies the second rule and it stops copying the labels. Notice that P is applied only to the children of the column nodes. This way we can add a simple rule for $\mathcal{Q}$ to prevent from having the same column numbers:

$$\exists x \quad x{\downarrow}y, x \sim y$$

We need also an additional predicate P_R that works like P but for rows. Every time the positive program changes the row it can add the intensional predicate P_R for the old row label to be carried under the node with the new row label. The difference is that P_R will not have a non-recursive rule when it meets a # node because we need to carry the row label to the end. Moreover it will need an additional rule like for the predicate U to go through the # node. The only non-recursive rule for $P_R(x)$ is when it finds the last tile:

$$P_R(x) \leftarrow \quad x{\uparrow}^3 y, k_F(y)$$

This concludes the proof. □

We sketch the intuition for "data trees" models. The idea is to use the same encoding. The problem is that we use constants for the tiles and \$, #. Let Σ be the set of these constants. We define the finite set of labels as $\overline{\Sigma} = \Sigma \cup \{\flat\}$, where $\flat$ is an additional symbol. The alphabet for the data trees is $\overline{\Sigma} \times \mathbb{D}$ where $\mathbb{D}$ is the infinite set of data. We write similar programs as in Proposition 8. For constants we use the same labels from Σ, for other nodes (rows and columns) we use $\flat$. The problem is that the relation $\sim$ does not compare the labels but their data. But our programs do not actually need to use $\sim$ to compare constants. We can explicitly write the labels and force them to be the same.

Proposition 9. *Containment of* $\mathsf{RTPQ}(\downarrow, \downarrow_+)$ *queries in* $\mathsf{UCQ}(\downarrow, \downarrow_+)$ *queries is* EXPSPACE-*hard over trees.*

Proof. We consider the satisfiability problem of $\varphi \wedge \neg\psi$, where $\varphi \in \mathsf{RTPQ}(\downarrow, \downarrow_+)$ and $\psi \in \mathsf{UCQ}(\downarrow, \downarrow_+)$. Let $\mathcal{M}$ be a deterministic Turing machine working on an exponential tape. Let n be a natural number. We may assume that $\mathcal{M}$ never uses more than 2^n tape cells. The query φ will code the run of the machine, and the query ψ will ensure its correctness. This encoding is similar to the encoding of the grid in Proposition 5. Here, instead of tiles we write the cell content: the state and the tape symbol. The rows represent the configurations and the columns the numbers of the tape cells.

In the proof of Proposition 5 we worked on a word structure and we could force consistent order of identifiers for columns. On trees it is not possible thus we use the binary counter for columns. The 0's and 1's from the counter might not be on one path but the negative query ensures that it works properly.

The run of the $\mathcal{M}$ is encoded in a word of the form $(R \cdot k_1 \cdot \dots \cdot k_n \cdot C)^*$ where R is a label representing the configuration, in $k_1, \dots, k_n$ we encode the number of the tape cell (one should think of the k_i's as bits) and C is the content of the cell (if the head is in this cell then the label contains also the state of the machine in this configuration). If there is an accepting run for $\mathcal{M}$ then this will be encoded in one of the paths of the tree model for $\varphi \wedge \neg\psi$.

Let Σ be the alphabet of $\mathcal{M}$ and Q the set of states. We assume that Σ includes a special letter $\flat$ to represent the empty cell. Our queries will use the alphabet $A = \Sigma \times (Q \cup \{\$\})$ to encode the content of the cells. The symbol \$ is for cells without of the head of the machine. We will also use two labels 0 and 1 to encode cell numbers.

We now define the query φ. We start from a subquery α which contains three consecutive tape cells in the same configuration and one in the next configuration:

$$\alpha^{C,C',C''} = \underbrace{(R)^{in}_{out}{\downarrow_{n+1}}C}_{\text{previous cell}}\downarrow\underbrace{R{\downarrow}k_1{\downarrow}\dots{\downarrow}k_n{\downarrow}C'}_{\text{current cell}}\downarrow\underbrace{R{\downarrow_{n+1}}C''}_{\text{next cell}}\downarrow_+$$
$$\underbrace{R{\downarrow}1{\downarrow}\dots{\downarrow}1}_{\text{last cell}}\downarrow_2 R'\downarrow_+\underbrace{R'{\downarrow}k_1{\downarrow}\dots{\downarrow}k_n{\downarrow}C'''}_{\text{current cell in next conf.}}$$

where $C, C', C'', C''' \in A$. The axis $\downarrow_2$ was used to omit the content of the last cell. Looking at the content of the previous, current, and next cell, we can determine the content C''' of the current cell in the next configuration. We will be taking disjunction of these queries over possible C, C', C''. We need special queries $\alpha_{beg}, \alpha_{end}, \alpha_{penul}$ to determine the content of the cell in the next configuration for the first, last and penultimate cell. These are extreme cases when there is no previous cell, or the last cell is also the next cell, etc. In these queries we know the precise values of the counter so we do not need the variables k_i. We show only how to write queries α_{end}:

$$\alpha^{C,C'}_{end} = \underbrace{(R)^{in}_{out}{\downarrow_{n+1}}C}_{\text{previous cell}}\downarrow\underbrace{R{\downarrow}1{\downarrow}\dots{\downarrow}1{\downarrow}C'}_{\text{current (last) cell}}\downarrow R'\downarrow_+\underbrace{R'{\downarrow}1{\downarrow}\dots{\downarrow}1{\downarrow}C''}_{\text{current cell in next conf.}}$$

We also need queries to encode the cell number. For every $1 \leq i \leq n$ and every $z \in \{0,1\}$ we define:

$$\beta^z_i = (R)^{in}_{out}{\downarrow_i}z$$

Using them we will ensure that between two consecutive tape cells there is a 0 or 1 on every level (not necessarily on the path between them). We know there is a unique number encoded in binary between two configurations. We want them to be consecutive (modulo 2^n). For every i we need patterns π_i like in Figure 4.

The β queries occur always directly after α queries and the π queries are after β queries so in future we use:

$$\overline{\alpha} = \alpha \cdot (\beta^0_1 + \beta^1_1) \cdot \ \dots \ \cdot (\beta^0_n + \beta^1_n) \cdot \bigcup_{1 \leq i \leq n} \pi_i$$

$$\begin{tikzcd}
 & & & (R)^{in} \arrow[dlll, "1"'] \arrow[dl, "i-1"'] \arrow[d, "i"] \arrow[dr, "i+1"] \arrow[drrr, "n"] \arrow[ddrrrr, bend left, "n+2"] & & & & & & & \\
k_1 & \cdots & k_{i-1} & 0 & 1 & \cdots & 1 & & & & \\
 & & & & & & & (R)_{out} \arrow[dlll, "1"'] \arrow[dl, "i-1"'] \arrow[d, "i"] \arrow[dr, "i+1"] \arrow[drrr, "n"] & & & \\
 & & & & k_1 & \cdots & k_{i-1} & 1 & 0 & \cdots & 0
\end{tikzcd}$$

Fig. 4: Pattern π_i.

for α being any of the previously defined patterns $\alpha^{C,C',C''}$, $\alpha^{C,C'}_{end}$, etc. In short we enforce n labels 1 or 0 below the configuration (β queries). In the end we check if the counter changes properly (π queries). Now we introduce the subqueries γ that move to the next configuration or go to the accepting state:

$$\gamma_{next} = (R)^{in}{\downarrow}1{\downarrow}\ldots{\downarrow}1{\downarrow_2}(R')_{out}$$
$$\gamma_{stay} = (R)^{in}{\downarrow}1{\downarrow}\ldots{\downarrow}1{\downarrow}(C_{acc})_{out}$$

We assume that there is only on accepting state that can occur only in the last cell of the tape. To represent a configuration we introduce the query φ_{conf}

$$\varphi_{conf} = \Big(\underbrace{\bigcup_{C,C'} \overline{\alpha}^{C,C'}_{beg}}_{\text{first cell}} \cdot \underbrace{\big(\bigcup_{C,C',C''} \overline{\alpha}^{C,C',C''} \big)^*}_{\text{next cells}} \cdot$$
$$\underbrace{\bigcup_{C,C',C''} \overline{\alpha}^{C,C',C''}_{penul}}_{\text{penultimate cell}} \cdot \underbrace{\bigcup_{C,C'} \overline{\alpha}^{C,C'}_{end}}_{\text{last cell}} \cdot \underbrace{(\gamma_{next} + \gamma_{stay})}_{\text{conf. change}} \Big)$$

We have to write the first configuration explicitly, its cells are empty. One can easily write a query φ_0 corresponding to it.

We finish when we are in the accepting state. We can detect it with a small query $(C_{acc})^{in}_{out}$, where $C_{acc} \in A$ is the label representing a cell with the head of the machine in the accepting state. Finally we define φ as:

$$\varphi = \varphi_0 \cdot (\varphi_{conf})^* \cdot (C_{acc})^{in}_{out}$$

Now we define the query ψ which detects the possible mistakes and makes the binary counter work. We start with the binary counter. The β queries enforced that between two configurations on every level there is a 0 or a 1. First, we assure that we do not have both at any level. We need a disjunction of the following patterns for every $1 \leq i \leq n$:

With similar patterns we can detect errors for α queries: the content of the tape cells could be not unique. The queries are defined in Figure 5.

$$\begin{tikzcd}
& x \arrow[dl, "i"'] \arrow[dr, "i"] & \\
0 & & 1
\end{tikzcd}$$

$$\begin{tikzcd}
& x \arrow[dl, "i"'] \arrow[dr, "i"] & \\
Q_1 & & Q_2
\end{tikzcd}$$

Fig. 5: For every $Q_1 \neq Q_2$ we add 3 queries for $i = n, i = 2n + 1$ and $i = 3n + 2$.

These queries ensure that the content of the cells in the current, previous and next configurations are unique and α queries work properly. To detect errors for the cell in the next configurations we need additionally queries from Figure 6. These queries look for the last cell in a configuration. Then they say that the next configuration cannot have different values in the cell encoded by k_i labels.

$$\begin{tikzcd}
& & & x \arrow[d, "+"'] \arrow[dr, "+"] & & & & & \\
& & & R \arrow[dlll, "1"'] \arrow[dl, "n"'] \arrow[d, "n+1"] & R \arrow[d, "1"] \arrow[drr, "n"] \arrow[drrr, "n+1"] & & & & \\
1 & \cdots & 1 & R' \arrow[d] & 1 & \cdots & 1 & R'' \arrow[d] & \\
& & & R' \arrow[dll, "1"'] \arrow[d, "n"] \arrow[dr, "n+1"] & & & & R'' \arrow[dll, "1"'] \arrow[d, "n"] \arrow[dr, "n+1"] & \\
& k_1 & \cdots & k_n & Q_1 & k_1 & \cdots & k_n & Q_2
\end{tikzcd}$$

Fig. 6: The patterns that find errors in the next configuration. We take the disjunction of these patterns for all $Q_1 \neq Q_2$, where $Q_i \in A$.

The next possible error can occur in the counter. In the pattern π_i we cannot ensure that k_i will be mapped to 0 or 1; there could be additional values in the tree. Thus we need a query that shows that two identical sequences k_i have the same number encoded in binary. For every i we write a pattern like in Figure 7.

When we have these patterns then we can think that k_i in π_i correspond to 0 or 1. The remaining errors such as having two heads in different cells in one configuration are easy to detect. $\square$

Proposition 10. *Containment for* L-Datalog($\downarrow$) *and* D-Datalog($\downarrow$) *is 2-*EXPTIME*-hard over trees and ranked trees.*

Proof. We show the proof for trees but the model we enforce with $\mathcal{P} \wedge \neg\mathcal{Q}$ is actually a binary tree. Thus the proof works for ranked trees. We start from the proof for

$$\begin{tikzcd}
 & & & x \arrow[d, "+"'] \arrow[dr, "+"] & & & & \\
 & & & R \arrow[dlll, "1"'] \arrow[dl, "n"'] \arrow[d, "i"] & R \arrow[d, "1"] \arrow[drr, "n"] \arrow[drrr, "i"] & & & \\
k_1 & \cdots & k_n & 0 & k_1 & \cdots & k_n & 1
\end{tikzcd}$$

Fig. 7: The patterns that ensure that the counter is correct. We take the disjunction of these for all $1 \leq i \leq n$.

L-Datalog($\downarrow$) programs. We encode a run of an AExpSpace Turing machine in the tree. Let M be an alternating Turing Machine with set of states Q and tape alphabet B. Let n be a number s.t. M uses at most 2^n tape cells on input of length n. We can assume that every configuration of M has at most two succesor configurations. First we show how to encode one configuration of M (state and contents of tape). We do so by enforcing a full binary tree of height n. For this we need an alphabet $S = \{0\} \cup \bigcup_{1 \leq i \leq n} \{(L, i), (R, i)\}$. The label 0 is for the root and $(L, i), (R, i)$ denote respectively the left and right son of the previous node. The tape is encoded in the leafs of this tree.

We will use symbols from B as constants but with additional marking if the head of M is on that cell, so we use symbols from $A = \{0, 1\} \times B$, Q and S. We store the state of M in the node below the node labeled with $\#$. Each state will be equipped with an extra marker L, R or $\exists$ to simplify the navigation in the tree. The marker $\exists$ means that this state is a successor of an existential state, and L, R denote successors of universal states.

We start from defining the rules for the positive program $\mathcal{P}$. The programs starts in a node labeled with $\top$ (unique in the tree, this can be enforced by the negative program). The goal rule is

$$G(X) \leftarrow \top(X), X{\downarrow}Y, \mathit{conf}_{(\exists, q_s)}(Y)$$

where q_s is the intial state. Our program will traverse the tree in infix order. For every state $q \in Q$ there are three rules

$$\begin{aligned}
\mathit{conf}_{(L,q)}(X) &\leftarrow (L, q)(X), X \downarrow Y, 0(Y), \mathit{downleft}^0(Y) \\
\mathit{conf}_{(R,q)}(X) &\leftarrow (R, q)(X), X \downarrow Y, 0(Y), \mathit{downleft}^0(Y) \\
\mathit{conf}_{(\exists,q)}(X) &\leftarrow (\exists, q)(X), X \downarrow Y, 0(Y), \mathit{downleft}^0(Y)
\end{aligned}$$

the symol L, R or $\exists$ denotes if the previous state was existential ($\exists$) or universal (L, R). Additionally L denotes the left of two configurations following an universal state and R denotes the right one. This ordering of configurations following an universal state is necessary to allow the Datalog programs to check the correctness of the encoding. Notice that we start traversing from the left node. We show additional rules for traversing.

$$\begin{aligned}
\textit{downleft}^i(X) &\leftarrow X \downarrow Y, (L, i+1)(Y), \textit{downleft}^{i+1}(Y)\\
&i = 0, \ldots, n-1\\
\textit{downleft}^n(X) &\leftarrow X \downarrow Y, \textit{store}(Y)\\
\textit{store}(X) &\leftarrow a(X), Y \downarrow X, (L, n)(Y), \textit{upleft}^n(X)\\
\textit{store}(X) &\leftarrow a(X), Y \downarrow X, (R, n)(Y), \textit{upright}^n(X)\\
&\text{for every symbol } a \in A\\
\textit{upleft}^i(X) &\leftarrow Y \downarrow X, \textit{downright}^{i-1}(Y)\\
&i = 1, \ldots, n\\
\textit{downright}^i(X) &\leftarrow X \downarrow Y, (R, i+1)(Y), \textit{downleft}^{i+1}(Y)\\
&i = 0, \ldots, n-1\\
\textit{upright}^i(X) &\leftarrow Y \downarrow X, (R, i-1)(Y), \textit{upright}^{i-1}(Y)\\
&i = 1, \ldots, n\\
\textit{upright}^i(X) &\leftarrow Y \downarrow X, (L, i-1)(Y), \textit{upleft}^{i-1}(Y)\\
&i = 2, \ldots, n
\end{aligned}$$

Notice that when we reach *downleft*^{n} we stop traversing the tree and the program uses the rule *store* to write the content of the tape. That is why there is no rule *downright*^{n}.

After traversing the whole configuration tree the program stops in *upright*^{0} and needs to go to the next configuration. This step can be done for any q' that can be the next state reachable by M from q.

$$\textit{upright}^0(X) \leftarrow Y \downarrow X, Y \downarrow Z, \#(Z), Z \downarrow Z', \textit{conf}_{(L,q')}(Z')$$
$$\textit{upright}^0(X) \leftarrow Y \downarrow X, Y \downarrow Z, \#(Z), Z \downarrow Z', \textit{conf}_{(\exists,q')}(Z')$$

The negative program can easily check if L and $\exists$ are used correctly (that is, the first rule should be used only if current state is universal and the second only if it is existential). Additionally, for each accepting state q:

$$\textit{upright}^0(X) \leftarrow Y \downarrow X, Y \downarrow Z, \#(Z), \textit{up}(Y), (L, q)(Y)$$
$$\textit{upright}^0(X) \leftarrow Y \downarrow X, Y \downarrow Z, \#(Z), \textit{up}(Y), (R, q)(Y)$$
$$\textit{upright}^0(X) \leftarrow Y \downarrow X, Y \downarrow Z, \#(Z), \textit{up}(Y), (\exists, q)(Y)$$

If in the predicate $up(X)$ the label of X is an accepting state, the rule makes the program go up the tree. If it is (L, q) then it needs to go down to the right configuration. Otherwise if it is $(\exists, q)$ or (R, q) then the rule for up is to simply go up.

$$\begin{aligned}
up(X) &\leftarrow (\exists, q)(X), Y \downarrow X, Y' \downarrow Y, Z \downarrow Y', up(Z)\\
up(X) &\leftarrow (R, q)(X), Y \downarrow X, Y' \downarrow Y, Z \downarrow Y', up(Z)\\
up(X) &\leftarrow (L, q)(X), Y \downarrow X, Y \downarrow Z, \textit{conf}_{(R,q')}(Z)\\
up(X) &\leftarrow \top(Y), Y \downarrow X
\end{aligned}$$

Notice that the last rule for up is the only non-recursive rule in the program.

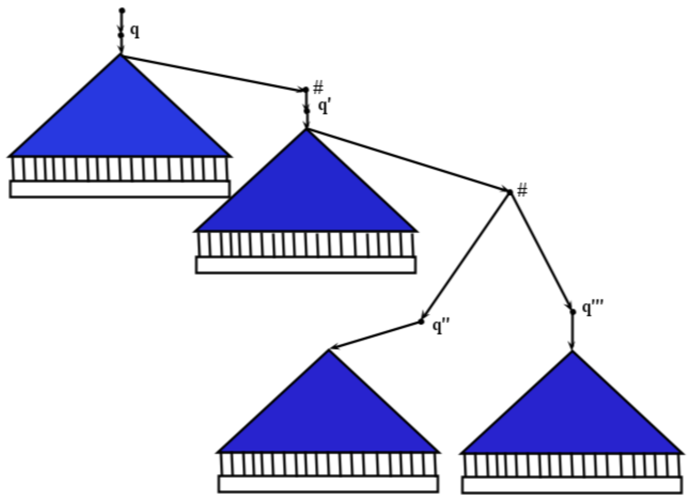


Fig. 8: Part of an encoding of a run of M in the tree. Each dark subtree is a full binary tree of height n. Configurations of M are encoded in the white nodes below the dark subtrees.

We briefly sketch the rules for the negative program $\mathcal{Q}$. They ensure that consecutive configurations are consistent with transitions of M. We need to prevent the following errors:

1. Two different tape values in one location.
2. Head of M in two different cells.
3. Symbol on tape changing incorrectly between two consecutive configurations.
4. State changing incorrectly between two consecutive configurations.
5. $\top$ in any node but the root of the tree.

We can easily write appropriate formulae because the tape alphabet is finite and independent of n and two nodes storing the value of the same cell in two consecutive configurations of M are always in distance $2n + 3$, so non-recursive queries can compare them. For example, the negation of $\exists x, y\ x \downarrow y, \top(y)$ enforces the last property.

We now briefly discuss the case for D-Datalog$(\downarrow)$ programs. These programs do not have the ability to go up the tree. Thus once we put an intensional predicate below the root of the configuration we will not be able to come back. Note that we needed this ability only in two cases – to build the full binary tree to keep a configuration; and to return to a configuration when there were two next configurations after a universal state. But now our programs do not have to be linear. Actually, both of these things are much more naturally expressed using the non-linearity of the program. Thus the encoding of the run of the machine will be the same but with minor changes in the rules. The binary tree can be created by rules like:

$$\begin{aligned}down^i(X) \leftarrow X \downarrow Y, X \downarrow Z, (L, i+1)(Y), (P, i+1)(Z),\\ down^{i+1}(Y), down^{i+1}(Z)\end{aligned}$$

For the rule $down^0(X)$ we need an additional intensional predicate on X because we cannot go back to it again and we need to add subtrees with new configurations.

Similarly for the universal states we can add two different successor roots for the next configurations with intensional predicates on both of them.

In this program the only non-recursive rules will be the rules that store the content of the tapes, i.e., $down^n$. $\square$