

Eliminating recursion from monadic datalog programs on trees

Filip Mazowiecki, Joanna Ochremiak, and Adam Witkowski

University of Warsaw

Abstract. We study the problem of eliminating recursion from monadic datalog programs on trees with an infinite set of labels. We show that the boundedness problem, i.e., determining whether a datalog program is equivalent to *some* nonrecursive one is undecidable but the decidability is regained if the descendant relation is disallowed. Under similar restrictions we obtain decidability of the problem of equivalence to a *given* nonrecursive program. We investigate the connection between these two problems in more detail.

1 Introduction

Among logics with fixpoint capabilities, one of the most prominent is datalog, which augments unions of conjunctive queries (positive existential first order formulae) with recursion. Datalog originated as a declarative programming language, but later found many applications in databases as a query language. The gain in expressive power does not, however, come for free. Compared to unions of conjunctive queries, evaluating a datalog program is harder [22] and basic properties such as containment or equivalence become undecidable [21].

Since the source of the difficulty in dealing with datalog programs is their recursive nature, the first line of attack in trying to optimize such programs is to eliminate the recursion. It is well-known that a nonrecursive datalog program can be rewritten as a union of conjunctive queries. The main focus of this paper is therefore the equivalence of recursive datalog programs to unions of conjunctive queries.

Example 1. The programs in this example work on databases that use binary predicates *likes* and *knows*, and a unary predicate *trendy*. First, consider the following pair of datalog programs:

$$\begin{array}{ll}
 \mathcal{P}_1 & \mathcal{P}'_1 \\
 buys(X, Y) \leftarrow likes(X, Y) & buys(X, Y) \leftarrow likes(X, Y) \\
 buys(X, Y) \leftarrow trendy(X), buys(Z, Y) & buys(X, Y) \leftarrow trendy(X), likes(Z, Y)
 \end{array}$$

The program \mathcal{P}_1 is recursive because its second rule refers to the predicate *buys*. It can be shown that \mathcal{P}_1 is equivalent to the nonrecursive program \mathcal{P}'_1 . Consider, on the other hand, the following pair of programs:

$$\begin{array}{ll}
\mathcal{P}_2 & \mathcal{P}'_2 \\
buys(X, Y) \leftarrow likes(X, Y) & buys(X, Y) \leftarrow likes(X, Y) \\
buys(X, Y) \leftarrow knows(X, Z), buys(Z, Y) & buys(X, Y) \leftarrow knows(X, Z), likes(Z, Y)
\end{array}$$

It can be shown that \mathcal{P}_2 is not equivalent to the nonrecursive program \mathcal{P}'_2 . Moreover, this program is not equivalent to any nonrecursive program.

The example above (taken from [18]) presents two approaches to eliminating recursion from datalog programs. Either we want to determine for a given datalog program if it is equivalent to *some* nonrecursive datalog program or decide whether a given datalog program is equivalent to a *given* nonrecursive program. These problems bear some similarities but in general they are separate. The latter is decidable [11], while the former, called the *boundedness* problem, is not [15,16].

Negative results for the full datalog fueled interest in its restrictions [4,6,7]. Important restrictions include *monadic* programs, using only unary predicates in the heads of rules; *linear* programs, with at most one use of an intensional predicate per rule; and *connected* programs, where within each rule all variables that are mentioned are connected to each other. Throughout this paper only monadic datalog programs are considered. In [12] Cosmadakis *et al.* show that for such programs the boundedness problem becomes decidable. Moreover, they use the same techniques to prove that the containment problem of two monadic datalog programs is decidable. These results suggest that under some additional assumptions the boundedness problem and the equivalence problem are more related.

In this paper we study connected, monadic datalog programs restricted to tree-structured databases. Our models are finite trees whose nodes carry labels from an infinite alphabet that can be tested for equality. Over such structures the problem of equivalence to a given union of conjunctive queries is known to be undecidable [1,17]. We show that the boundedness problem is also undecidable. In some cases, however, we regain decidability of both problems in the absence of the descendant relation. On ranked trees we show that the equivalence and the boundedness problems become decidable (in 2-EXPTIME). On unranked trees we prove that the equivalence of a linear program to a non-recursive one is EXPSpace-complete. We finish with an analysis of the connection between the equivalence and the boundedness problems and show that under some assumptions they are equi-decidable.

Organization. In Section 2 we introduce datalog programs and some basic definitions. In Section 3 we deal with the problem of equivalence to a given non-recursive datalog program. In Section 4 we analyze the boundedness problem. Finally, in Section 5 we explore the connection between the two approaches to eliminating recursion from datalog programs and show that under some assumptions the arising decision problems are equi-decidable. We conclude in Section 6 with possible directions for future research. Due to the page limit most of the proofs are moved to the appendix.

2 Preliminaries

In this paper we work over finite trees labeled with letters from an infinite alphabet Σ . The trees are unranked by default, but we also work with ranked trees, in particular with words. We use the standard notation for axes: \downarrow, \downarrow_+ stand, respectively, for child and descendant relations. We assume that each node has one label. A binary relation \sim holds between nodes with identical labels and there is a unary predicate a for each $a \in \Sigma$, holding for the nodes labeled with a .

We begin with a brief description of the syntax and semantics of datalog; for more details see [2] or [8]. A **datalog program** \mathcal{P} over a relational signature S is a finite 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. All variables in the body that are not used in the head are implicitly quantified existentially. The **size of a rule** is the number of different variables that appear in it.

The relational symbols, or predicates, in S fall into two categories. **Extensional** predicates are the ones explicitly stored in the database; they are never used in the heads of rules. In our setting they come from $\{\downarrow, \downarrow_+, \sim\} \cup \Sigma$. The alphabet Σ is infinite, but the program \mathcal{P} uses only its finite subset which we denote by $\Sigma_{\mathcal{P}}$. **Intensional** predicates, used both in the heads and bodies, are defined by the rules.

The program is evaluated by generating all atoms (over intensional predicates) that can be inferred from the underlying structure (tree) by applying the rules repeatedly, to the point of saturation. Each inferred atom can be witnessed by a **proof tree**: an atom inferred by a rule r from intensional atoms A_1, A_2, \dots, A_n is witnessed by a proof tree with the root labeled by r , and n children which are the roots of the proof trees for atoms A_i (if r has no intensional predicates in its body then the root has no children).

There is a designated predicate called the **goal** of the program. We will often identify the goal predicate with the program, i.e., we write $\mathcal{P}(X)$ if the goal predicate of the program \mathcal{P} holds on the node X . When evaluated in a given database D , the program \mathcal{P} results in the unary relation $\mathcal{P}(D) = \{X \in D \mid \text{such that } \mathcal{P}(X) \text{ holds}\}$. If $\mathcal{P}(D) \subseteq \mathcal{Q}(D)$ for every database D then we say that the program \mathcal{P} is **contained** in the program \mathcal{Q} . If the containment holds both ways then the programs \mathcal{P} and \mathcal{Q} are **equivalent**.

Example 2. The program below computes the nodes from which one can reach some label a along a path where each node has a child with identical label and a descendant with label b (or has label b itself).

| | | |
|--|-------------------|--|
| $P(X) \leftarrow X \downarrow Y, P(Y), X \downarrow Y', X \sim Y', Q(X)$ | (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) | |

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

The notion of proof trees comes from papers on datalog over general structures (see e.g. [11]). As shown in Example 2 proof trees illustrate how the program evaluates. While on general structures for a given proof tree one can always find a model such that the proof tree witnesses a correct evaluation of the program, on tree structures this is not so simple. One reason is that we allow only one label for every node. As a result, rules like $P(X) \leftarrow a(X), b(X)$ cannot be satisfied for $a \neq b$. Moreover, nodes have a unique father. Because of this it is not easy to determine whether a given proof tree is a witness of an evaluation of the program on some model and it does not suffice to eliminate unsatisfiable rules. Proof trees for which such a model exists will be called **satisfiable proof trees**.

Example 3. The program below goes down a tree along a path labeled with a . Then it goes up the tree until it finds a node labeled with b .

| | | | |
|--|---------|--------------|--------------|
| $P(X) \leftarrow X \downarrow Y, a(Y), P(Y)$ | (p_3) | p_3 | p_3 |
| | | \downarrow | \downarrow |
| $P(X) \leftarrow Q(X)$ | (p_4) | p_4 | p_3 |
| | | \downarrow | \downarrow |
| $Q(X) \leftarrow Y \downarrow X, Q(Y)$ | (q_3) | q_3 | p_4 |
| | | \downarrow | \downarrow |
| $Q(X) \leftarrow b(X)$ | (q_4) | q_4 | q_3 |
| | | \downarrow | \downarrow |
| | | q_4 | q_4 |

The first proof tree is satisfiable, but the second proof tree is not satisfiable because it enforces both labels a and b on the same node.

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 sake of simplicity: one can always turn a 0-ary predicate Q to a unary predicate $Q(X)$ by introducing a dummy variable X . For connected programs (described below) this restriction matters.

For a datalog rule r , let G_r be a graph whose vertices are the variables used in r and an edge is placed between X and Y if the body of r contains an atomic formula $X \downarrow Y$ or $X \downarrow_+ Y$. In G_r we distinguish a **head node** and **intensional nodes**. The latter are all variables from the body of r used by intensional predicates. A program \mathcal{P} is **connected** if for each rule $r \in \mathcal{P}$, the graph G_r is connected¹.

Previous work on datalog on arbitrary structures often considered the case of connected programs [12,15]. The practical reason is that real-life programs tend to be connected (cf. [3]). Also, rules which are not connected combine pieces of

¹ One could consider a definition allowing additionally nodes connected by the equality relation but we expect that this would be as hard as the disconnected case e.g. the main problem we leave open in Section 3, the equivalence of child-only non-linear programs, becomes undecidable by the results of [17] for boolean queries.

unrelated data, corresponding to the *cross product*, an unnatural operation in the database context. It seems even more natural to assume connectedness when working with tree-structured databases. We shall do so. We write $\text{Datalog}(\downarrow, \downarrow_+)$ for the class of connected monadic datalog programs, and $\text{Datalog}(\downarrow)$ for connected monadic programs that do not use the relation \downarrow_+ .

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 linear programs we shall use the letter **L**, e.g., $\text{L-Datalog}(\downarrow)$ means linear programs from $\text{Datalog}(\downarrow)$. The program from Example 2 is connected, but not linear. The program from Example 3 is both connected and linear.

Conjunctive queries (CQs) are existential first order formulae of the form $\exists x_1 \dots x_k \varphi$, where φ is a conjunction of atoms. We will consider **unions of conjunctive queries** (UCQs), corresponding to nonrecursive programs with a single intensional predicate (goal) which is never used in the bodies of rules. Since UCQs can be seen as datalog programs, we can speak of connected UCQs and as for datalog, we shall always assume connectedness. We denote the classes of connected queries by $\text{CQ}(\downarrow, \downarrow_+)$, $\text{CQ}(\downarrow)$, $\text{UCQ}(\downarrow, \downarrow_+)$, $\text{UCQ}(\downarrow)$, respectively.

3 Equivalence

For datalog programs the containment problem can be reduced to the equivalence problem. Let \mathcal{P} be a datalog program and let \mathcal{Q} be a UCQ. Then $\mathcal{P} \subseteq \mathcal{Q}$ iff $\mathcal{P} \vee \mathcal{Q} \equiv \mathcal{Q}$. Notice that this reduction does not depend on the type of the programs (e.g., disallowing \downarrow_+ relation; or assuming linearity) but relies on the fact that datalog programs are closed under the disjunction.

The containment problem for datalog programs has been studied on trees in other contexts [1,5,14,17]. In [17] containment of datalog programs in UCQs on data trees was analyzed in detail for boolean queries, which are queries that return the answer 'yes' if they are satisfied in some node of a given database, and the answer 'no', otherwise. More formally, a datalog program \mathcal{P} defines a *boolean query* $\mathcal{P}_{\text{Bool}}(D)$ which equals 1 iff $\mathcal{P}(D)$ is nonempty and 0 otherwise.

The containment problem is usually solved by considering the dual problem. For unary queries, it is the question whether there exist a database D and $X \in D$ such that $\mathcal{P}(X)$ and $\neg \mathcal{Q}(X)$, where $\neg \mathcal{Q} = D \setminus \mathcal{Q}(D)$. For boolean queries, it is the question if there exist a database D and $X, Y \in D$ such that $\mathcal{P}(X)$ and $\neg \mathcal{Q}(Y)$. For datalog programs over trees, if we allow the \downarrow_+ relation this distinction does not make much of a difference (intuitively because using \downarrow_+ one can move from a node X to any node Y). Thus a closer look at the proofs of Theorem 1 and Proposition 3 from [17] gives the following.

Proposition 4. *Over ranked and unranked trees the containment problem of L-Datalog(\downarrow, \downarrow_+) programs in UCQ(\downarrow, \downarrow_+) is undecidable.*

In the rest of this section we work only with fragments of datalog without the \downarrow_+ relation. We start with ranked trees.

Theorem 5. *The containment problem is 2-EXPTIME-complete for Datalog(\downarrow) over ranked trees. In the special case of words it is PSPACE-complete.*

The above result yields tight complexity bounds for the equivalence problem of Datalog(\downarrow) programs to UCQ(\downarrow) programs over ranked trees. To prove Theorem 5 (see Appendices B.1 and B.2) we define automata that simulate the behavior of datalog programs, modifying the approach of [17]. The new construction gives better complexity results for non-linear programs².

In the rest of this section we focus on the equivalence problem of Datalog(\downarrow) programs to UCQ(\downarrow) programs over unranked trees. For the containment problem, this question was left open in [1].

For boolean queries, the containment problem of Datalog(\downarrow) programs in UCQ(\downarrow) programs was proved undecidable in [17]. Decidability was restored for the linear fragment, for which it was shown to be 2-EXPTIME-complete. We improve the complexity for unary queries using different techniques (see Appendices B.4 and B.5).

Theorem 6. *The containment problem of an L-Datalog(\downarrow) program in a UCQ(\downarrow) program is EXPSpace-complete over unranked trees.*

Unfortunately our approach does not generalize to the non-linear case. On the other hand, the proof of undecidability provided in [17] also cannot be adapted to work in our setting³. We leave the question of the decidability of containment for non-linear programs as an open problem.

The following lemma is proved in Appendix B.6 (we do not assume linearity).

Lemma 7. *The containment problem of UCQ(\downarrow) queries in Datalog(\downarrow) is in NPTime over ranked and unranked trees.*

As a corollary of Theorem 6 and Lemma 7 we obtain the main result of this section. The lower bound is carried from the containment problem.

Theorem 8. *The equivalence problem of an L-Datalog(\downarrow) program to a UCQ(\downarrow) program is EXPSpace-complete over unranked trees.*

4 Boundedness

Consider a datalog program \mathcal{P} with a goal predicate P . By $\mathcal{P}^i(D)$ we denote the collection of facts about the predicate P that can be deduced from a database D by at most i applications of the rules in \mathcal{P} . More formally, $\mathcal{P}^i(D)$ is the subset

² In [17] the non-linear case required an additional exponential blow-up. However, the improvement of complexity is not caused by considering unary instead of boolean queries. It is easy to see that Theorem 5 holds also in the boolean case.

³ Indeed, the main idea of the undecidability proof is to use the UCQ \mathcal{Q} to find errors in the run of a Turing machine encoded by the program \mathcal{P} . If the nonrecursive query \mathcal{Q} is unary it can only find errors close to the node X , such that $\mathcal{P}(X)$.

of $\mathcal{P}(D)$ derived using proof trees of height at most i , where the *height* of a tree is the length of the longest path from its root to a leaf. Then obviously

$$\mathcal{P}(D) = \bigcup_{i \geq 0} \mathcal{P}^i(D).$$

We say that the program \mathcal{P} is **bounded** if there exists a number n , depending only on \mathcal{P} , such that for any database D , we have $\mathcal{P}(D) = \mathcal{P}^n(D)$. Intuitively this means that the depth of recursion is independent of the input database⁴.

Each proof tree corresponds to a conjunctive query in a natural way. Therefore, we can always translate a datalog program to an equivalent, but possibly infinite, union of conjunctive queries. If the program is bounded then it is equivalent to a finite subunion of its corresponding conjunctive queries. For full datalog it is known that the opposite implication is also true, i.e., a program is bounded iff it is equivalent to a (finite) UCQ [19]. The same holds for the class $\text{Datalog}(\downarrow)$:

Proposition 9. *Let $\mathcal{P} \in \text{Datalog}(\downarrow)$. Then \mathcal{P} is bounded iff it is equivalent to a union of conjunctive queries $\mathcal{Q} \in \text{UCQ}(\downarrow)$.*

We remark that the above characterization (which we prove in Appendix C) is based on the existence of so-called *canonical databases* for CQs (see e.g. [10]) in $\text{Datalog}(\downarrow)$. The following example shows that without canonical databases equivalence to some UCQ does not necessarily imply boundedness. It relies on the fact that \downarrow_+ is the transitive closure of \downarrow .

Example 10. The program $\mathcal{P} \in \text{Datalog}(\downarrow, \downarrow_+)$ on the left is not bounded – finding b in a tree can take arbitrarily long. The program \mathcal{P}' on the right is a UCQ equivalent to \mathcal{P} .

$$\begin{array}{l} \mathcal{P} \\ P(X) \leftarrow X \downarrow_+ Y, a(Y) \\ P(X) \leftarrow X \downarrow Y, Q(Y) \\ Q(X) \leftarrow X \downarrow Y, Q(Y) \\ Q(X) \leftarrow b(X) \end{array} \quad \mathcal{P}' \quad \begin{array}{l} P(X) \leftarrow X \downarrow_+ Y, a(Y) \\ P(X) \leftarrow X \downarrow_+ Y, b(Y) \end{array}$$

We obtain a negative result for $\text{L-Datalog}(\downarrow, \downarrow_+)$ (see Appendix C.1).

Theorem 11. *The boundedness problem for $\text{L-Datalog}(\downarrow, \downarrow_+)$ is undecidable over words and ranked or unranked trees.*

In the following we work with fragments of datalog without the \downarrow_+ relation. For decidability results we use the automaton-theoretic approach of [12].

Theorem 12. *The boundedness problem for $\text{Datalog}(\downarrow)$ over words is in PSPACE.*

⁴ Observe that we are only interested in the output on the goal predicate. This is why the property we consider is sometimes called the *predicate* boundedness [16].

In the case of trees the same technique can be applied but the complexity increases (see Appendix C.2).

Theorem 13. *The boundedness problem for $\text{Datalog}(\downarrow)$ over ranked trees is in 2-EXPTIME .*

Over words, the relations \downarrow and \downarrow_+ are interpreted as the “next position” and the “following position”. Let X be a position in a word w . The n -neighbourhood of X in w is an infix of w , which begins on position $\max(1, X - n)$ and ends on position $\min(|w|, X + n)$. The following lemma is motivated by Proposition 3.2 of [12]. Its proof is provided in Appendix C.2.

Lemma 14. *Let \mathcal{P} be a $\text{Datalog}(\downarrow)$ program. Then \mathcal{P} is bounded iff there exists $n > 0$ such that for every word w and position X if $X \in \mathcal{P}(w)$ then $X \in \mathcal{P}(v)$, where v is the n -neighbourhood of X in w .*

Proof (of Theorem 12). A word w such that for some position X in w we have $X \in \mathcal{P}(w)$ but $X \notin \mathcal{P}(v)$, where v is the n -neighbourhood of X in w will be called an n -witness. By Lemma 14 a $\text{Datalog}(\downarrow)$ program \mathcal{P} is unbounded iff there exist n -witnesses for arbitrarily big $n > 0$.

Consider a $\text{Datalog}(\downarrow)$ program \mathcal{P} . Let Σ_0 be an alphabet that contains the set of labels used explicitly in the rules of \mathcal{P} together with N “fresh” labels, where N is the size of the biggest rule in \mathcal{P} . It is known [17] (and easy to verify) that any word w can be relabeled so that the obtained word w' uses only labels from Σ_0 , and for each position X we have that $X \in \mathcal{P}(w)$ iff $X \in \mathcal{P}(w')$. This is also true with respect to infixes, i.e., for every infix v of w , and every position X it holds that $X \in \mathcal{P}(v)$ iff $X \in \mathcal{P}(v')$, where v' is the corresponding infix of w' . Hence, we can verify the existence of n -witnesses over the finite alphabet Σ_0 .

In the proof of Theorem 5 (see Appendix B.1) a nondeterministic automaton is introduced that recognizes words over the alphabet Σ_0 satisfying \mathcal{P} . More precisely, the constructed automaton $\mathcal{A}_{\mathcal{P}}$ works over the alphabet $\Sigma_0 \times \{0, 1\}$, and accepts a word w iff it has exactly one position X marked with 1 such that $X \in \mathcal{P}(w)$. We denote the language recognized by $\mathcal{A}_{\mathcal{P}}$ by $L(\mathcal{A}_{\mathcal{P}})$. The size of this automaton is exponential in the size of \mathcal{P} .

Similarly, we obtain an automaton $\mathcal{N}_{\mathcal{P}}$ recognizing these words over the alphabet $\Sigma_0 \times \{0, 1\}$ which have exactly one position marked with 1 but do not belong to $L(\mathcal{A}_{\mathcal{P}})$. The size of $\mathcal{N}_{\mathcal{P}}$ is also exponential in the size of \mathcal{P} (there is no exponential blow up because the constructions in Appendix B.1 go through alternating automata) and the language it recognizes will be denoted $L(\mathcal{N}_{\mathcal{P}})$. Note that this language is closed under infixes containing the marked position.

We define a nondeterministic automaton $\mathcal{B}_{\mathcal{P}}$ which accepts exactly those words belonging to $L(\mathcal{A}_{\mathcal{P}})$ which have an infix that belongs to $L(\mathcal{N}_{\mathcal{P}})$. The states and transitions of $\mathcal{B}_{\mathcal{P}}$ are the states and transitions of the product automaton $\mathcal{A}_{\mathcal{P}} \times \mathcal{N}_{\mathcal{P}}$ together with the states and transitions of two copies of the automaton $\mathcal{A}_{\mathcal{P}}$ denoted $\mathcal{A}_{\mathcal{P}}^1$ and $\mathcal{A}_{\mathcal{P}}^2$. Let q_{init} be the initial state of $\mathcal{N}_{\mathcal{P}}$. For each state q of $\mathcal{A}_{\mathcal{P}}^1$ we add to $\mathcal{B}_{\mathcal{P}}$ an epsilon transition from the state q to the state (q, q_{init}) of the product automaton. Now, let F be the set of final states of $\mathcal{N}_{\mathcal{P}}$. For each

state q of $\mathcal{A}_{\mathcal{P}}^2$ and each $q_{fin} \in F$ we add to $\mathcal{B}_{\mathcal{P}}$ an epsilon transition from the state (q, q_{fin}) to q . The initial state of $\mathcal{B}_{\mathcal{P}}$ is the initial state of $\mathcal{A}_{\mathcal{P}}^1$ and the final states of $\mathcal{B}_{\mathcal{P}}$ are the final states of $\mathcal{A}_{\mathcal{P}}^2$. Hence, an accepting run of the automaton $\mathcal{B}_{\mathcal{P}}$ starts in $\mathcal{A}_{\mathcal{P}}^1$, moves to the product automaton at some point, reads an infix that belongs to $L(\mathcal{N}_{\mathcal{P}})$ and finally goes to $\mathcal{A}_{\mathcal{P}}^2$ to accept.

Let N be the number of states of the product automaton $\mathcal{A}_{\mathcal{P}} \times \mathcal{N}_{\mathcal{P}}$ plus 1. Suppose that $\mathcal{B}_{\mathcal{P}}$ accepts an N -witness w . Then, due to the pumping lemma, it accepts n -witnesses for arbitrarily big $n > 0$. To end the proof show that checking whether $\mathcal{B}_{\mathcal{P}}$ accepts some N -witness is in NLOGSPACE in the size of the automata $\mathcal{A}_{\mathcal{P}}$ and $\mathcal{N}_{\mathcal{P}}$ (i.e., in PSPACE in the size of \mathcal{P}).

An N -witness is a word that belongs to $L(\mathcal{A}_{\mathcal{P}})$ but the N -neighbourhood of the position marked with 1 belongs to $L(\mathcal{N}_{\mathcal{P}})$. The NLOGSPACE algorithm simulates a run of the automaton $\mathcal{B}_{\mathcal{P}}$. The size of $\mathcal{B}_{\mathcal{P}}$ is exponential in the size of \mathcal{P} but its states and transitions can be generated on the fly in polynomial space. The algorithm guesses a state from the $\mathcal{A}_{\mathcal{P}} \times \mathcal{N}_{\mathcal{P}}$ part and checks if it is reachable from the initial state. This is a simple reachability test which is in NLOGSPACE. Then it guesses some run of the $\mathcal{A}_{\mathcal{P}} \times \mathcal{N}_{\mathcal{P}}$ part, counts the number of transitions done before the one marked with 1, and ensures that it is at least N . After the transition marked with 1 it ensures that the automaton makes at least N more transitions before leaving the $\mathcal{A}_{\mathcal{P}} \times \mathcal{N}_{\mathcal{P}}$ part. For both of these counting procedures we need $\log(N)$ tape cells. Finally, the algorithm performs a second reachability test to check if the automaton can reach a final state.

There are three possible ways of how an N -witness v may look like. For simplicity, the algorithm described above does not deal with the case when the N -neighbourhood that belongs to $L(\mathcal{N}_{\mathcal{P}})$ is shorter than $2N + 1$ (which can happen if it begins at the first position of w or ends at the last position of w). Those possibilities can be verified similarly. \square

Notice that if \mathcal{P} is bounded then N from the proof above is the bound on the depth of recursion. Since the size of the constructed automaton is exponential in the size of the program \mathcal{P} , the UCQ which is equivalent to this program consists of proof trees of size at most exponential in the size of \mathcal{P} .

5 Boundedness vs equivalence

In this section we focus on the similarities between the boundedness and the equivalence problem for datalog programs. In Sections 3 and 4 those problems are treated separately but with similar techniques. Also in [12], where boundedness and equivalence are considered for monadic programs on arbitrary structures, both problems are solved using the same automata-theoretic construction. For these reasons we investigate the connection between the two problems in more detail. In contrast to the previous sections, in this section the structures under consideration are not necessarily trees or words.

Definition 15. *A class \mathcal{C} of datalog programs over a fixed class of databases is called **well-behaved** if:*

1. for every program $\mathcal{P} \in \mathcal{C}$ all the UCQs corresponding to the proof trees for \mathcal{P} belong to \mathcal{C} ,
2. containment of a UCQ in a datalog program is decidable for \mathcal{C} .

Condition (1) is satisfied for most natural classes of programs. In particular by the class of all datalog programs on arbitrary structures and the class $\text{Datalog}(\downarrow)$ on trees. For the class of datalog programs on arbitrary structures Condition (2) is also known to hold true (see [9,13,20]). Lemma 7 shows that the class $\text{Datalog}(\downarrow)$ on trees satisfies Condition (2). Hence both those classes are well-behaved.

We say that \mathcal{C} has a **computable bound** if there exists a computable function f such that if a datalog program \mathcal{P} in \mathcal{C} is bounded and $f(\mathcal{P}) = n$ then $\mathcal{P}(D) = \mathcal{P}^n(D)$ for any database D , i.e., for bounded programs the function f returns a bound on the depth of recursion. For programs which are not bounded f returns some arbitrary natural numbers.

Example 16. Consider the full datalog. It follows from the results of [12] that the class of monadic datalog programs on arbitrary structures has a computable bound. It is not stated explicitly but a closer analysis of the proofs gives that for a bounded program \mathcal{P} the depth of recursion can be bounded polynomially in the size of the automaton constructed to check if \mathcal{P} is bounded. For example, for a linear connected program the size of such an automaton is bounded exponentially in the size of the program.

The following theorem for a well-behaved class \mathcal{C} with a computable bound establishes a connection between the problems of boundedness and equivalence to a given UCQ.

Theorem 17. *For any well-behaved class \mathcal{C} with a computable bound the following conditions are equivalent:*

1. boundedness is decidable,
2. it is decidable whether two programs are equivalent, given that one of them is a UCQ.

Proof. Let f be the function from the definition of the computable bound. For the implication from (1) to (2), take programs \mathcal{P} and \mathcal{Q} which belong to \mathcal{C} and assume that \mathcal{Q} is a UCQ. Since \mathcal{C} is well-behaved, we only need to show how to decide whether \mathcal{P} is contained in \mathcal{Q} . It follows from the assumption that we can verify if \mathcal{P} is bounded. If this is the case, then let $f(\mathcal{P}) = n$. Observe that \mathcal{P} is equivalent to the UCQ \mathcal{P}' that corresponds to the proof trees for \mathcal{P} of height at most n . It remains to decide whether the UCQ \mathcal{P}' is contained in \mathcal{Q} .

Suppose now that \mathcal{P} is not bounded and consider a union \mathcal{R} of the programs \mathcal{P} and \mathcal{Q} . More formally, let \mathcal{R} be a program containing the rules of both programs \mathcal{P} and \mathcal{Q} . If the predicate \mathcal{Q} occurs in the program \mathcal{P} we rename it so that the predicates do not coincide. The goal predicate \mathcal{R} holds for X iff we have $\mathcal{P}(X)$ or $\mathcal{Q}(X)$. For this we introduce two additional rules $\mathcal{R}(X) \leftarrow \mathcal{P}(X)$ and $\mathcal{R}(X) \leftarrow \mathcal{Q}(X)$. The atoms $\mathcal{Q}(X)$ are all inferred in one step. Therefore, if \mathcal{R} is

unbounded then there exists X satisfying $\mathcal{P}(X)$ such that $\mathcal{Q}(X)$ does not hold, and hence \mathcal{P} is not contained in \mathcal{Q} . If \mathcal{R} is bounded then using f we construct an equivalent UCQ \mathcal{R}' and check whether it is equivalent to \mathcal{Q} . If this is the case then \mathcal{P} is contained in \mathcal{Q} . Otherwise it is not.

For the other implication, consider a datalog program $\mathcal{P} \in \mathcal{C}$ and let $f(\mathcal{P}) = n$. Then \mathcal{P} is bounded iff $\mathcal{P}(D) = \mathcal{P}^n(D)$ for any database D . Let \mathcal{Q} be the UCQ that corresponds to the proof trees of \mathcal{P} of height at most n . It suffices to decide whether the programs \mathcal{P} and \mathcal{Q} are equivalent. But this is decidable from the assumption that \mathcal{C} is well-behaved. \square

While assuming that a class of programs is well-behaved is natural, the existence of a computable bound is a strong assumption. It is needed since an algorithm that solves the boundedness problem might not be constructive, meaning that we do not know how big the equivalent UCQ is. However, deciding if such a function exists is usually as hard as solving the boundedness problem. From Example 16 we know that for monadic programs on arbitrary structures there exist constructive algorithms for the boundedness problem, and hence we have a computable bound. On the other hand, the undecidability results of the boundedness problem for datalog on arbitrary structures rely heavily on the fact that such a computable bound does not exist. In [15,16] the authors present reductions from the halting problem for 2-counter machines and Turing machines. If a datalog program is bounded then the size of the equivalent UCQ corresponds to the length of an accepting run of these machines, which of course cannot be bounded by a computable function. The results of our paper are, in this sense, similar: the positive results provide computable bounds whereas the negative results rely on the fact that such a function does not exist. For these reasons we conjecture that for well-behaved classes of datalog programs the decidability of the boundedness problem is equivalent to the decidability of finding a computable bound. If this conjecture holds true then Theorem 17 becomes an implication from (1) to (2) because the opposite implication is trivially satisfied.

6 Conclusions

The equivalence to a given nonrecursive program and the boundedness problem for $\text{Datalog}(\downarrow, \downarrow_+)$ are undecidable. To regain decidability we considered programs that do not use the \downarrow_+ relation. We showed that equivalence to a given UCQ over ranked trees is decidable, and over unranked trees it is decidable in the case of linear programs. We also showed the decidability of boundedness on words and ranked trees. In the most general case of non-linear $\text{Datalog}(\downarrow)$ programs over unranked trees we do not know if the two problems under consideration are decidable and we leave these questions as open problems.

We also investigated the connection between the boundedness and the equivalence to a UCQ. We showed that these problems are equivalently decidable for classes of programs with a computable bound. We suspect, however, that the existence of a computable bound for a class of programs is equivalent to the decidability of the boundedness problem. We also leave this as an open problem.

References

1. Serge Abiteboul, Pierre Bourhis, Anca Muscholl, and Zhilin Wu. Recursive queries on trees and data trees. In *ICDT*, pages 93–104, 2013.
2. Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison Wesley, 1995.
3. François Bancilhon and Raghu Ramakrishnan. An amateur’s introduction to recursive query processing strategies. In *ACM SIGMOD*, pages 16–52, 1986.
4. Michael Benedikt, Pierre Bourhis, and Pierre Senellart. Monadic datalog containment. In *ICALP*, pages 79–91, 2012.
5. Mikołaj Bojańczyk, Filip Murlak, and Adam Witkowski. Containment of monadic datalog programs via bounded clique-width. Accepted for ICALP, 2015.
6. Piero A. Bonatti. On the decidability of containment of recursive datalog queries - preliminary report. In *PODS*, pages 297–306, 2004.
7. Diego Calvanese, Giuseppe De Giacomo, and Moshe Y. Vardi. Decidable containment of recursive queries. *Theor. Comput. Sci.*, 336(1):33–56, 2005.
8. Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic programming and databases*. Springer-Verlag New York, Inc., 1990.
9. Ashok K. Chandra, Harry R. Lewis, and Johann A. Makowsky. Embedded implicational dependencies and their inference problem. In *STOC*, pages 342–354, 1981.
10. Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90, 1977.
11. Surajit Chaudhuri and Moshe Y. Vardi. On the equivalence of recursive and non-recursive datalog programs. In *PODS*, pages 55–66, 1992.
12. Stavros S. Cosmadakis, Haim Gaifman, Paris C. Kanellakis, and Moshe Y. Vardi. Decidable optimization problems for database logic programs (preliminary report). In *STOC*, pages 477–490, 1988.
13. Stavros S. Cosmadakis and Paris C. Kanellakis. Parallel evaluation of recursive rule queries. In *PODS*, pages 280–293, 1986.
14. André Frochoux, Martin Grohe, and Nicole Schweikardt. Monadic datalog containment on trees. In *Proceedings of the 8th Alberto Mendelzon Workshop on Foundations of Data Management*, 2014.
15. Haim Gaifman, Harry G. Mairson, Yehoshua Sagiv, and Moshe Y. Vardi. Undecidable optimization problems for database logic programs. *J. ACM*, 40(3):683–713, 1993.
16. Gerd G. Hillebrand, Paris C. Kanellakis, Harry G. Mairson, and Moshe Y. Vardi. Undecidable boundedness problems for datalog programs. *J. Log. Program.*, 25(2):163–190, 1995.
17. Filip Mazowiecki, Filip Murlak, and Adam Witkowski. Monadic datalog and regular tree pattern queries. In *MFCS*, pages 426–437, 2014.
18. Jeffrey F. Naughton. Data independent recursion in deductive databases. *J. Comput. Syst. Sci.*, 38(2):259–289, 1989.
19. Jeffrey F. Naughton and Yehoshua Sagiv. A simple characterization of uniform boundedness for a class of recursions. *J. Log. Program.*, 10(34):233 – 253, 1991.
20. Yehoshua Sagiv. Optimizing datalog programs. In *Foundations of Deductive Databases and Logic Programming.*, pages 659–698. Morgan Kaufmann, 1988.
21. Oded Shmueli. Equivalence of datalog queries is undecidable. *J. Log. Program.*, 15(3):231–241, 1993.
22. Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In *STOC*, pages 137–146, 1982.

A Definitions

A.1 Automata

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

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

As an intermediate automata model, closer to datalog than the bottom-up automata, we shall use the two-way alternating automata introduced in [12]. A **two-way alternating automaton** $\mathcal{A} = \langle \Gamma, Q, q_I, \delta \rangle$ consists of an alphabet Γ , a finite set of states Q , an initial state $q_I \in Q$, and a transition function

$$\delta: Q \times \Gamma \rightarrow \text{BC}^+(Q \times \{-1, 0, 1\})$$

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

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

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

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

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

The computation model of two-way alternating automata is very similar to that of datalog programs, making them a perfect intermediate formalism on the

road to nondeterministic bottom-up automata. From there one continues thanks to the following fact.

Proposition 18 ([12]). *Given a two-way alternating automaton \mathcal{A} (interpreted over words or ranked trees), one can compute (in time polynomial in the size of the input and output) single-exponential nondeterministic bottom-up automata recognizing the language $L(\mathcal{A})$ and its complement, respectively.*

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

A.2 Canonical models and homomorphisms

Let r be a satisfiable rule of a datalog program \mathcal{P} . Recall from Section 2 that G_r is a graph of nodes from r . A **pattern** π_r has the same nodes and edges as G_r but the type of edge between nodes (\downarrow or \downarrow_+) is distinguished. The nodes are labeled with variable names. If there is an extensional unary predicate, e.g. $a(X)$, specified by the rule then we replace the label X with a . We simulate the relation \sim by repeating variable labels.

Since in our setting the relation \downarrow_+ is disallowed, we can always transform a satisfiable rule r into an equivalent rule r' such that $\pi_{r'}$ is a tree. This is because our models are trees and therefore nodes that have a common child can be merged into one node.

Example 19. The rule P is transformed into its tree version P' . On the right there are patterns corresponding to these rules. The repeated occurrence of X represents the relation \sim in the patterns.

$$\begin{array}{ccc}
 P(X) \leftarrow X \downarrow Y, Y \downarrow Z, T \downarrow Z, a(T), X \sim Z & \begin{array}{c} X \\ \swarrow \\ Y \quad a \\ \searrow \swarrow \\ X \end{array} & \begin{array}{c} X \\ \downarrow \\ a \\ \downarrow \\ X \end{array} \\
 \Downarrow & & \rightsquigarrow \\
 P'(X) \leftarrow X \downarrow Y, Y \downarrow Z, a(Y), X \sim Z & &
 \end{array}$$

A **homomorphism** from a pattern π_r to a model tree t is a function between nodes that preserves the extensional predicates. A proof tree is witnessing an evaluation of the program on a given model t iff for all rules there is a homomorphism from their patterns to t such that the intensional nodes are mapped to the same nodes as the head nodes in the following rules. The connection between patterns and datalog is explained in more detail in [17].

From a satisfiable proof tree we obtain a **canonical model**. First we change rules to patterns and merge head nodes with intensional nodes. Nodes labeled with variables are relabeled with fresh labels (preserving the equalities forced by \sim). The obtained graph can be seen as a pattern of the proof tree. Then we turn it into a tree similarly as in Example 19.

It is easy to see that it suffices to consider the containment problem only on canonical models. If there is a model t for $\mathcal{P} \wedge \neg \mathcal{Q}$ then there is a witnessing proof tree for \mathcal{P} on t . The canonical model corresponding to this proof tree is also a model for $\mathcal{P} \wedge \neg \mathcal{Q}$.

B Equivalence

We decide containment by constructing an automaton that is non-empty iff there is a counterexample to containment. To do this, we mark a single node in a tree, and use the automaton to verify if the goal predicates of programs in question are satisfied in this node. Formally, we extend the alphabet by taking its product with $\{0, 1\}$, and recognize models which have exactly one node marked with 1. To obtain tight complexity bounds, we use two-way alternating automata. The same technique was used in [14].

B.1 Special case: words

Over words, the relations \downarrow and \downarrow_+ are interpreted as the “next position” and the “following position”.

Lemma 20. *Let $\mathcal{P} \in \text{Datalog}(\downarrow)$ and let Σ_0 be a finite alphabet. There exists a two-way alternating automaton that accepts all words over $\Sigma_0 \times \{0, 1\}$ with exactly one position with label $(a, 1)$ for some $a \in \Sigma_0$, such that \mathcal{P} holds in that position. The automaton can be constructed in time polynomial in $|\mathcal{P}|$ and $|\Sigma_0|$.*

Proof. Let us fix a program $\mathcal{P} \in \text{Datalog}(\downarrow)$ and a finite alphabet Σ_0 . The alphabet is $\Sigma_0 \times \{0, 1\}$ but most of the time the second component is ignored. Since we work over words (and consider only connected programs) without loss of generality we can assume that each rule r is of the form

$$H(x_0) \leftarrow \bigwedge_{i=k}^{\ell-1} x_i \downarrow x_{i+1} \wedge \psi(x_k, x_{k+1}, \dots, x_\ell),$$

where $k \leq 0 \leq \ell$ and $\psi(x_k, x_{k+1}, \dots, x_\ell)$ is a conjunction of atoms over unary predicates and \sim ; that is, it does not use \downarrow . This means that the pattern corresponding to the body of r is a word.

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

$$Q \times \{-N, -N + 1, \dots, N\}$$

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

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

$$\Sigma_0 \cup \mathcal{P} \cup \{q_0\};$$

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

$$\begin{aligned}\delta(q_0, (-, 0)) &= (q_0, +1), \\ \delta(q_0, (-, 1)) &= (r_{\text{goal}}, 0),\end{aligned}$$

where r_{goal} is the goal rule of \mathcal{P} . This is the only case when $\mathcal{A}_{\mathcal{P}}$ does not ignore the component $\{0, 1\}$ in the alphabet. That is we require that there is 1 in the second component when the first goal rule is applied. When we are in state $r \in \mathcal{P}$, regardless of the current letter, we check that the body of r can be matched in the input word in such a way that x_0 is mapped to the current position:

$$\delta(r, -) = \bigwedge_{a(x_i)} (a, i) \wedge \bigwedge_{x_i \sim x_j} \bigvee_{b \in \Sigma_0} (b, i) \wedge (b, j) \wedge \bigwedge_{R(x_i)} \bigvee_{r' \in \mathcal{P}_R} (r', i),$$

where $a(x_i)$, $x_i \sim x_j$, and $R(x_i)$ range respectively over labels, \sim , and intensional atoms of r , and $\mathcal{P}_R \subseteq \mathcal{P}$ is the set of rules defining intensional predicate R . In state $a \in \Sigma_0$ we simply check that the letter in the current position is a :

$$\delta(a, a) = \top, \quad \text{and} \quad \delta(a, b) = \perp \text{ for } b \neq a.$$

Checking correctness and the size bounds for $\mathcal{A}_{\mathcal{P}}$ poses no difficulties. Taking a product of $\mathcal{A}_{\mathcal{P}}$ with an automaton (of size linear in $|\Sigma_0|$) that checks if there is exactly one position with label $(a, 1)$ for some $a \in \Sigma_0$ gives the automaton from the statement.

Now we can show the proof of Theorem 5 for the case of words.

Proof (of Theorem 5 (words)). In Proposition 2 of [17] it is shown that over words it suffices to check satisfiability of $\mathcal{P} \wedge \neg \mathcal{Q}$ over an alphabet Σ_0 of linear size. For programs \mathcal{P} and \mathcal{Q} , let $\mathcal{A}_{\mathcal{P}}$ and $\mathcal{A}_{\mathcal{Q}}$ be alternating two-way automata given by Lemma 20. From automata $\mathcal{A}_{\mathcal{P}}$ and $\mathcal{A}_{\mathcal{Q}}$, by Proposition 18, we obtain one-way non-deterministic automata $\mathcal{B}_{\mathcal{P}}$ and $\mathcal{B}_{\neg \mathcal{Q}}$ of exponential size that recognize respectively the language $L(\mathcal{A}_{\mathcal{P}})$ and the complement of $L(\mathcal{A}_{\mathcal{Q}})$. From this we easily get a product automaton $\mathcal{B}_{\mathcal{P} \wedge \neg \mathcal{Q}}$ equivalent to the query $\mathcal{P}(x) \wedge \neg \mathcal{Q}(x)$. Indeed, it accepts all words over Σ_0 with exactly one position x marked with 1, such that $\mathcal{P}(x) \wedge \neg \mathcal{Q}(x)$.

The size of $\mathcal{B}_{\mathcal{P} \wedge \neg \mathcal{Q}}$ is exponential in the size of \mathcal{P}, \mathcal{Q} , but its states and transitions can be generated on the fly in polynomial space. To check emptiness of $\mathcal{B}_{\mathcal{P} \wedge \neg \mathcal{Q}}$ we make a simple reachability test, which is in NLOGSPACE. Altogether, this gives a PSPACE algorithm. \square

B.2 Ranked trees

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

Lemma 21. *Let $\mathcal{P} \in \text{Datalog}(\downarrow)$ be a program with rules of size at most n and let Σ_0 be a finite alphabet. There exists a two-way alternating automaton $\mathcal{A}_{\mathcal{P}}$ of size $\mathcal{O}(\|\mathcal{P}\| \cdot |\Sigma_0|^n \cdot n)$ recognizing trees over $\Sigma_0 \times \{0, 1\}$ with exactly one node with label $(a, 1)$ for some $a \in \Sigma_0$, such that \mathcal{P} holds in that node.*

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

Since we are working on trees we can further transform the program so that the patterns corresponding to the rules of the program are trees (with *in* and *out* nodes positioned arbitrarily). Indeed, it can be done by unifying variables x and y whenever the rule contains $x \downarrow z$ and $y \downarrow z$ for some variable z , and removing rules containing atom $u \downarrow u$, or atoms $a(u)$ and $b(u)$ for some variable u and distinct letters a and b (see Example 19). This modification does not increase the size of the program.

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

$$H(x_0) \leftarrow a(x_0) \wedge \bigwedge_{i=1}^{\ell} \text{ax}_i(x_0, x_i) \wedge \psi(x_0, x_1, \dots, x_{\ell})$$

where $a \in \Sigma_0$, $\text{ax}_i(x_0, x_i)$ is either $x_0 \downarrow x_i$ or $x_i \downarrow x_0$, and $\psi(x_0, x_1, \dots, x_{\ell})$ is a conjunction of (monadic) intensional atoms. That is, one rule can only test the label and some intensional predicates for the current node, and demand existence of neighbours (children or parents) satisfying some intensional predicates. This modification introduces auxiliary intensional predicates, but the size of the program increases only by $\mathcal{O}(n)$ factor.

The resulting program is essentially a two-way alternating automaton \mathcal{B} , only given in a different syntax. The automaton from the statement is obtained by modifying the automaton \mathcal{B} similarly as in the case of words.

Proof (of Theorem 5 (trees)). In Theorem 1 of [17] it is shown that for trees it suffices to verify containment over a finite alphabet Σ_0 , although for trees Σ_0 is of exponential size. Using Lemma 21 and Proposition 18 we reduce the containment problem to the emptiness problem for a nondeterministic tree automaton of a double exponential size in $|\mathcal{P}|$, and test emptiness with the standard PTIME algorithm. \square

The lower bounds can be obtained by a straightforward modifications of the results in [17].

B.3 Satisfiability on unranked trees

Proposition 22. *The satisfiability problem for L-Datalog(\downarrow) on unranked trees is in EXPTIME.*

Before proving this result let us introduce the notation.

Definition 23. Let Σ_0 be a finite alphabet. A **universal Σ_0 -tree** is a full $|\Sigma_0|$ -ary tree over Σ_0 such that every non-leaf node has a child with each label from Σ_0 . For $a \in \Sigma_0, n \in \mathbb{N}$, we will denote by U_n^a a universal Σ_0 -tree of height n and with a in the root.

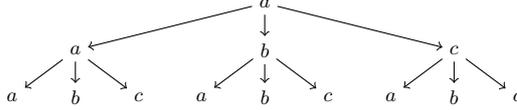


Fig. 1: A universal Σ_0 -tree U_2^a for $\Sigma_0 = \{a, b, c\}$

The proof will proceed as follows. First, we will show that if \mathcal{P} is satisfiable, then it is satisfiable in a universal Σ_0 -tree. Then it is easy to see (combining Lemma 21 and Proposition 18) that the set of universal Σ_0 -trees satisfying \mathcal{P} is regular and recognized by an automaton with number of states double exponential in $|\mathcal{P}|$. For linear programs however, we can do better and get an EXPTIME algorithm.

Lemma 24. Let $\mathcal{P} \in \text{Datalog}(\downarrow)$ and let Σ_0 be a finite set of labels, s.t. $\Sigma_{\mathcal{P}} \subseteq \Sigma_0$. The program \mathcal{P} is satisfiable iff \mathcal{P} is satisfiable in a universal Σ_0 -tree.

Proof. It suffices to show that if \mathcal{P} is satisfiable, then it is satisfied in some universal Σ_0 -tree. The other direction is obvious. Let t be a model for \mathcal{P} . Recall that $\Sigma_{\mathcal{P}}$ is the set of constants used in \mathcal{P} . First, we can change all labels from t that are not in Σ_0 to a single label chosen from Σ_0 (preserving the equalities). Since $\text{Datalog}(\downarrow)$ programs do not use negation and \neq this operation can only make the set $\mathcal{P}(t)$ bigger. Next, we perform the following operation. If a node of v has two or more children with the same labels then we merge these children into one node v . The resulting node has children from both of the merged nodes. It is easy to check that this operation preserves homomorphisms and does not change the emptiness of the set $\mathcal{P}(t)$. We apply this procedure until there are no siblings with the same label. Finally we add nodes to the obtained tree so that it becomes a universal Σ_0 -tree. Of course adding nodes cannot decrease the set $\mathcal{P}(t)$, which finishes the proof. \square

From now on we assume that \mathcal{P} is a linear program. We will actually prove a stronger result that will be useful for deciding the containment of a datalog program in a UCQ. We will show an algorithm for calculating all possible ways of evaluating the program \mathcal{P} in the universal Σ_0 -tree such that the evaluation uses the root of this tree.

First, we need to introduce a notion of a partial matching of a datalog program. We say that a rule is matched to a tree t if there is a homomorphism from

its pattern into t . Let $r_1 r_2 \dots r_n$ be a proof word. A **partial matching** m of a program \mathcal{P} into a tree t is an infix $r_i \dots r_j$ of a proof word such that all the rules r_{i+1}, \dots, r_{j-1} are matched completely and r_i and r_j are partially matched, such that the images of the intensional nodes are equal to the following head nodes.

Each partial matching m can be represented by a pair of partial homomorphisms from the patterns of the first and the last rule of the infix of the proof word. We are interested in the partial matchings that map one of the nodes of the pattern to the root of the tree. Thus each partial homomorphism can be represented as a partial function from pattern π into Σ_0 . Of course there are also partial matchings with nodes mapped below the root of the tree, and one end of a partial matching may be not possible to extend. This situation can arise when the goal rule is at the beginning of the matching; or the non-recursive rules are in the last position (leaves). We use an additional symbol OK to mark this situation.

We denote the set of all partial matchings of \mathcal{P} by $Match(\mathcal{P})$. The size of $Match(\mathcal{P})$ is exponential in the size of \mathcal{P} . Obviously it suffices to calculate the set of all partial matchings into a tree to determine if it satisfies \mathcal{P} .

Lemma 25. *Let Σ_0 be a finite alphabet. The set of partial matchings of \mathcal{P} matched in a root of any Σ_0 -universal tree can be calculated in time exponential in $|\mathcal{P}|$ for linear programs.*

Proof. For a tree t we will denote the set of partial matchings in the root of t by $matched(t)$. Observe that because every partial matching of \mathcal{P} in U_{n-1}^a is also a partial matching of \mathcal{P} in U_n^a then $matched$ is monotonic, i.e., $matched(U_{n-1}^a) \subseteq matched(U_n^a)$. This observation yields a simple algorithm. There are $|\Sigma_0|$ different universal trees of height n . To calculate $matched(U_n^a)$ for each $a \in \Sigma_0$ it suffices to join partial matchings from $\bigcup_{b \in \Sigma_0} matched(U_{n-1}^b)$ using the root node labeled with a and add the previously calculated $matched(U_{n-1}^a)$. Note that if $matched(U_n^a) = matched(U_{n+1}^a)$, then $matched(U_n^a) = matched(U_m^a)$ for all $m > n$. Therefore, the described procedure requires at most $|Match(\mathcal{P})|$ steps to terminate, each step takes $O(|Match(\mathcal{P})|)$ time which gives an EXPTIME algorithm. \square

B.4 Proof of the upper bound in Theorem 6

Let \mathcal{P} be a L-Datalog(\downarrow) program and let \mathcal{Q} be a UCQ(\downarrow). Our goal is to determine whether for all databases D we have $\mathcal{P}(D) \subseteq \mathcal{Q}(D)$. We solve the dual problem and look for a counterexample for the containment, i.e., a database D and a node $X \in D$ such that $X \in \mathcal{P}(D)$ but $X \notin \mathcal{Q}(D)$. Moreover, we can assume that D is a canonical model. Let n be the size of the biggest conjunct in \mathcal{Q} . Since \mathcal{Q} is nonrecursive and connected, to determine if $X \in \mathcal{Q}(D)$ it suffices to check the subtree of D containing nodes of distance at most n from X .

We shall refer to \mathcal{P} as the positive query and to \mathcal{Q} as the negative query. We define an automaton $\mathcal{A} = (Q, A, \delta, q_0, F)$ that essentially recognizes satisfiable proof words for \mathcal{P} simultaneously checking if the negative query is satisfied on

the canonical model of the read word. The alphabet $A = \{r_1, \dots, r_m\}$ is the set of rules of the program \mathcal{P} .

We define the set of states Q as a cartesian product of three components, i.e., $Q = Q_1 \times Q_2 \times Q_3$. We describe each component separately. Recall that $\Sigma_{\mathcal{P}}$ denotes the set of constants used explicitly in rules of program \mathcal{P} . Let N be the size of the biggest rule in \mathcal{P} . Let B_1 be an alphabet of N different letters and let B_2 be an alphabet of $2n + 1$ letters, disjoint from B_1 .

In the first component Q_1 the automaton stores a labeled pattern corresponding to the currently read letter (rule). Formally,

$$Q_1 = \sum_{r \in A} (\Sigma_{\mathcal{P}} \cup B_1 \cup B_2)^{\pi_r}.$$

We identify the pattern π_r with the set of its nodes, thus Q_1 is the set of patterns, whose nodes are labeled with elements of the set $\Sigma_{\mathcal{P}} \cup B_1 \cup B_2$. The intended meaning of B_1 and B_2 will be explained later.

In the second component Q_2 the automaton stores a word w of length at most $2n + 1$ and its position compared to the node X . Formally

$$Q_2 = \sum_{\substack{1 \leq i \leq 2n+1 \\ 0 \leq k, l \leq n}} (\Sigma_{\mathcal{P}} \cup B_1 \cup B_2)^{[i]} \times (k, l),$$

where $[i] = \{1, \dots, i\}$ and $(\Sigma_{\mathcal{P}} \cup B_1 \cup B_2)^{[i]}$ is the set of words of length i with labels from $\Sigma_{\mathcal{P}} \cup B_1 \cup B_2$. This word is a representation of an ancestor-path starting from the intensional node v_i of the current pattern stored in Q_1 . This is necessary to verify if the proof word is satisfiable. The ancestor path could be arbitrary long but, as we will see, we only need to remember nodes that are of distance at most n from X (there are at most $2n + 1$ such nodes). Additionally the automaton remembers how this path lays compared to X . For this it stores a pair of numbers (k, l) such that $0 \leq k, l \leq n$. Let v_a be the least common ancestor of X and v_i . The number k denotes the distance between X and v_a , and the number l denotes the distance between v_a and v_i . Note that $k + l$ is the distance between v_i and X . Also if $k = 0$ then v_i is a descendant of X , and if $l = 0$ then X is a descendant of v_i .

The last component Q_3 is the set of partial homomorphisms of the patterns corresponding to CQs from the negative query \mathcal{Q} . Let w be the word stored in the second component and let $A_{\mathcal{Q}}$ be the set of all conjuncts φ from \mathcal{Q} . Formally, $Q_3 = \sum_{\varphi \in A_{\mathcal{Q}}} F_{\varphi}$ where F_{φ} is the set of all partial functions from π_{φ} to $\Sigma_{\mathcal{P}} \cup B_1 \cup B_2 \cup \{b\} \cup \{w_1, \dots, w_{|w|}\}$. The interpretation of the labels will be explained later.

We now define the transition relation δ . Suppose that the automaton reads a new letter r . Let $q = (q_1, q_2, q_3) \in Q_1 \times Q_2 \times Q_3$ be the previous state. We show how the automaton calculates its new state $q' = (q'_1, q'_2, q'_3)$.

In the first component the automaton starts from checking if the rule r is proper for the intensional predicate in the previous rule; or if it is the first letter then the automaton checks if it is the goal predicate. If none of these cases

holds then the automaton immediately rejects the word. Otherwise it labels π_r in two phases. In the first phase it labels its head node v_h with the same label that the intensional node in q_1 has. Also the labels of the nodes that are ancestors of v_h must match the corresponding labels from the path in q_2 . Then the automaton labels nodes that have an explicit label from $\Sigma_{\mathcal{P}}$. In the second phase the automaton guesses the remaining labels from $\Sigma_{\mathcal{P}} \cup B_1 \cup B_2$ respecting the \sim relation. If there is a node on which \sim forces two different labels, then the automaton rejects the word. This way we use a small alphabet to represent an arbitrary large set of labels. If in the state q' we use a label that is also used in the state q but \sim does not force them to be the same then we assume that in the canonical model they are different labels.

In the second component the automaton updates first the pair (k, l) so that it agrees with the location of the new intensional node with respect to X . Then it creates a new ancestor-path whose labels have to agree with the labels of the old path in q_2 , and the labels of those nodes in q'_1 that are ancestors of v_i . The case when the distance of the new intensional node to X is bigger than n is explained later.

In the last component the automaton starts from updating the old partial functions. All labels that appeared in q_1 but were not used in the first phase are replaced with b . The intended meaning is that these labels no longer appear in the model. Actually this is where we use the crucial feature of the canonical models. Since we use fresh labels whenever it is possible the automaton can forget all labels that will no longer appear.

The automaton forgets all partial homomorphisms that have unmapped nodes such that their label is forced by \sim to be equal to a node labelled by b . This is because such homomorphisms can never be fulfilled. Then the automaton extends the remaining homomorphisms with new nodes from q'_1 . The label w_i denotes the fact that the node was mapped to the corresponding node from the path in q'_2 . The next step is to relabel the partial homomorphisms so that they agree with the new path. This way the automaton knows where it can extend the homomorphisms. Note that if there is a partial homomorphism without any w_i then it can be discarded because it cannot be extended. If at any time one of the homomorphisms becomes a full homomorphism then the automaton rejects the word.

So far we explained the behavior for the letters in the proof word that have the intensional node of distance at most n from X . This is of course not the only possible case, but we already noticed that nodes of bigger distance have no impact on the negative query. Because of this now we can use the results for the satisfiability problem. Suppose that the automaton reads a letter r such that its intensional node is of distance bigger than n from X . The automaton updates the third component of its state in the usual way and rejects the word if a full homomorphism is found. Let v be the ancestor of the intensional node in r such that v is of distance n from X . The automaton assumes that there is a universal tree t_v over the alphabet $\Sigma_{\mathcal{P}} \cup B_1 \cup B_2$ (see Definition 23) below v . It calculates the set $matched(t_v)$ and finds all matchings that have the rule r as the first rule

with the node v in the root. The automaton chooses one of the matchings but the last rule r' can also have the intensional node below v . Then it proceeds with r' as it did with r . Eventually the automaton guesses a matching such that the intensional node of the last rule r'' is of distance at most n from X . Then it stores r'' in q'_1 and updates the other states in the usual way. If instead of the last rule there is OK then the automaton accepts the word.

Notice that the node v could not exist. This happens when the least common ancestor of X and the intensional node of r is of distance bigger than n from X . If such a situation occurs then, since we assumed that we work on canonical models, all nodes from the next rules will be of distance bigger than n from X . Thus it suffices to check satisfiability starting from the rule r .

We slightly modified the canonical models using universal trees. For the positive program we showed in Lemma 24 that we can use universal trees; and for the negative program we assured that the changes are on nodes that are of distance bigger than n from X .

The constructed automaton is non-empty iff there is a canonical model for $\mathcal{P} \wedge \neg\mathcal{Q}$. We need to bound the size of the set of states. In the first component every labelled rule $(B_1 \cup B_2 \cup \Sigma_{\mathcal{P}})^{\pi_r}$ is of exponential size in $|\mathcal{P}|$ and the number of rules is bounded by the size of \mathcal{P} . The second component is a set of triples: two numbers and a word of size at most $2n+1$, which is exponential in the size of \mathcal{P} , \mathcal{Q} . The third component is the powerset of all partial homomorphisms which is double exponential in the size of \mathcal{P} and \mathcal{Q} . Thus the whole automaton is bounded double exponentially. However, its states and transitions can be generated on the fly in exponential space. To check its emptiness we make a simple reachability test, which is in NLOGSPACE. We use the results about satisfiability to generate all transitions, but by Proposition 22 this can be done in EXPTIME. Altogether, this gives an algorithm in EXPSPACE.

B.5 Proof of the lower bound in Theorem 6

We consider the satisfiability problem of $\mathcal{P} \wedge \neg\mathcal{Q}$, where $\mathcal{P} \in \text{L-Datalog}(\downarrow)$ and $\mathcal{Q} \in \text{UCQ}(\downarrow)$. To prove hardness, for a number n and a Turing machine M , we construct datalog programs \mathcal{P} and \mathcal{Q} of size polynomial in $|M|$ and n such that $\mathcal{P} \wedge \neg\mathcal{Q}$ is satisfiable iff M accepts the empty word using not more than 2^n tape cells. The program \mathcal{P} will encode the run of the machine, and the program \mathcal{Q} will ensure its correctness.

Assume that B is the tape alphabet of M , Q is the set of states, F is the set of accepting states and δ is the transition relation. The finite alphabet used by the programs will contain sets B and $B \times Q$. The symbols from $B \times Q$ will be used to mark the position of the head on the tape and the state of the machine.

We now define the rules of the positive program \mathcal{P} . The program starts in a node labeled with \top . We encode each configuration of M (the current state and the tape contents) by enforcing a full binary tree of height n . For this we need the alphabet $\Sigma_{\mathcal{P}}$ to contain the set $\sum_{1 \leq i \leq n} \{(L, i), (R, i)\}$. The predicates (L, i) and (R, i) denote the left and right son of the previous node, respectively. The tape is encoded in the nodes below the leaves of the tree. The label of the node above

the root of the tree is used as an identifier of the encoded configuration. We will refer to it as an *identification node*.

The goal rule is

$$G_{\mathcal{P}}(X) \leftarrow \top(X), X \downarrow Y, \text{Init}(Y), Y \downarrow Z, \text{conf}(Z).$$

It means that the encoding of the initial configuration of the machine, which identification node is labeled with *Init*, is stored in the tree (note that *Init* belongs to $\Sigma_{\mathcal{P}}$). The program will then traverse the configuration trees one by one in an infix order.

$$\begin{aligned} \text{conf}(X) &\leftarrow X \downarrow Y, (L, 1)(Y), \text{downleft}^1(Y) \\ \text{downleft}^i(X) &\leftarrow X \downarrow Y, (L, i + 1)(Y), \text{downleft}^{i+1}(Y) \\ &\quad i = 1, \dots, n - 1 \\ \text{downleft}^n(X) &\leftarrow X \downarrow Y, \text{store}(Y) \\ \text{store}(X) &\leftarrow a(X), Y \downarrow X, (L, n)(Y), \text{upleft}^n(X) \\ \text{store}(X) &\leftarrow a(X), Y \downarrow X, (R, n)(Y), \text{upright}^n(X) \\ &\quad \text{for every symbol } a \in B \cup (B \times Q) \\ \text{upleft}^i(X) &\leftarrow Y \downarrow X, \text{downright}^{i-1}(Y) \\ &\quad i = 1, \dots, n \\ \text{downright}^i(X) &\leftarrow X \downarrow Y, (R, i + 1)(Y), \text{downleft}^{i+1}(Y) \\ &\quad i = 0, \dots, n - 1 \\ \text{upright}^i(X) &\leftarrow Y \downarrow X, (R, i - 1)(Y), \text{upright}^{i-1}(Y) \\ &\quad i = 2, \dots, n \\ \text{upright}^i(X) &\leftarrow Y \downarrow X, (L, i - 1)(Y), \text{upleft}^{i-1}(Y) \\ &\quad i = 2, \dots, n \\ \text{upright}^1(X) &\leftarrow Y \downarrow X, \text{next}(Y). \end{aligned}$$

Observe 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 .

The program finishes traversing the tree in *next* and goes to the next configuration of the machine. We ensure that the identification node of the next configuration has the same label as the root of the tree which encodes the previous one. This will enable the negative program to check the correctness of the encoding.

$$\text{next}(X) \leftarrow Y \downarrow X, Z \downarrow Y, Z \downarrow \hat{Y}, \hat{Y} \sim X, \hat{Y} \downarrow \hat{X}, \text{conf}(\hat{X}).$$

We finish when we find an accepting state. That is for every letter $a \in B$ and every $q \in F$ we have two non-recursive rules

$$\begin{aligned} \text{store}(X) &\leftarrow (a, q)(X), Y \downarrow X, (L, n)(Y) \\ \text{store}(X) &\leftarrow (a, q)(X), Y \downarrow X, (R, n)(Y). \end{aligned}$$

Now let us define the rules of the negative program \mathcal{Q} , which will be a disjunction of queries describing possible errors in the encoding. The content of the tape has to be defined uniquely. Hence, for each pair of different symbols a and b from $B \cup (B \times Q)$ we have a rule

$$G_{\mathcal{Q}}(X) \leftarrow \top(X), X \downarrow Y, Y \downarrow X_0, X_0 \downarrow X_1, X_1 \downarrow X_2, \dots, X_n \downarrow Z_1, X_n \downarrow Z_2, a(Z_1), b(Z_2).$$

We cannot ensure that each configuration tree has its identification node labeled differently, but we can guarantee that trees with the same labels of the identification nodes encode the same configurations. For each pair of different symbols a and b from $B \cup (B \times Q)$ we introduce a rule

$$G_{\mathcal{Q}}(X) \leftarrow \top(X), X \downarrow Y, X \downarrow \hat{Y}, Y \sim \hat{Y}, Y \downarrow X_0, \hat{Y} \downarrow \hat{X}_0, X_0 \downarrow X_1, \hat{X}_0 \downarrow \hat{X}_1, X_1 \sim \hat{X}_1, \dots, \\ \dots, X_{n-1} \downarrow X_n, \hat{X}_{n-1} \downarrow \hat{X}_n, X_n \sim \hat{X}_n, X_n \downarrow Z, \hat{X}_n \downarrow \hat{Z}, a(Z), b(\hat{Z}).$$

We can also easily enforce that the configuration tree labeled with *Init* encodes the initial configuration of the machine with an empty word stored on the tape.

Finally we have to make sure that the way the positive program \mathcal{P} moves from one configuration to another is consistent with the transition function of the machine. To do this we consider changes in the content of any three consecutive tape cells, i.e., we take all tuples $(a_1, a_2, a_3, b_1, b_2, b_3)$ of symbols from $B \cup (B \times Q)$, such that: if a_1, a_2, a_3 encode a content of three consecutive tape cells $i, i+1, i+2$, respectively, then it is not possible for the machine to have b_1, b_2, b_3 on those positions in the next configuration. For each of those tuples there is a set of $2(n-1)$ rules in \mathcal{Q} . The rules are constructed depending on the least common ancestor of the three leafs which encode the consecutive tape cells. We write them down for $n=3$. There are two rules that deal with the case when the least

common ancestor is the root of the tree

$$\begin{aligned}
G_{\mathcal{Q}}(X) \leftarrow & \top(X), X \downarrow Y, Y \downarrow X_0, X \downarrow \hat{Y}, X_0 \sim \hat{Y}, \\
& X_0 \downarrow X_1 \downarrow X_2 \downarrow X_3, X_0 \downarrow X'_1 \downarrow X'_2 \downarrow X'_3, X_0 \downarrow X''_1 \downarrow X''_2 \downarrow X''_3, \\
& (L, 1)(X_1), (R, 2)(X_2), (L, 3)(X_3), \\
& (L, 1)(X'_1), (R, 2)(X'_2), (R, 3)(X'_3), \\
& (R, 1)(X''_1), (L, 2)(X''_2), (L, 3)(X''_3), \\
& Y \downarrow \hat{X}_0, \hat{X}_0 \downarrow \hat{X}_1 \downarrow \hat{X}_2 \downarrow \hat{X}_3, \hat{X}_0 \downarrow \hat{X}'_1 \downarrow \hat{X}'_2 \downarrow \hat{X}'_3, \hat{X}_0 \downarrow \hat{X}''_1 \downarrow \hat{X}''_2 \downarrow \hat{X}''_3, \\
& X_1 \sim \hat{X}_1, X'_1 \sim \hat{X}'_1, \dots, X''_3 \sim \hat{X}''_3, \\
& X_3 \downarrow Z_1, a_1(Z_1), X'_3 \downarrow Z_2, a_2(Z_2), X''_3 \downarrow Z_3, a_3(Z_3), \\
& \hat{X}_3 \downarrow \hat{Z}_1, b_1(\hat{Z}_1), \hat{X}'_3 \downarrow \hat{Z}_2, b_2(\hat{Z}_2), \hat{X}''_3 \downarrow \hat{Z}_3, b_3(\hat{Z}_3)
\end{aligned}$$

$$\begin{aligned}
G_{\mathcal{Q}}(X) \leftarrow & \top(X), X \downarrow Y, Y \downarrow X_0, X \downarrow \hat{Y}, X_0 \sim \hat{Y}, \\
& X_0 \downarrow X_1 \downarrow X_2 \downarrow X_3, X_0 \downarrow X'_1 \downarrow X'_2 \downarrow X'_3, X_0 \downarrow X''_1 \downarrow X''_2 \downarrow X''_3, \\
& (L, 1)(X_1), (R, 2)(X_2), (R, 3)(X_3), \\
& (R, 1)(X'_1), (L, 2)(X'_2), (L, 3)(X'_3), \\
& (R, 1)(X''_1), (L, 2)(X''_2), (R, 3)(X''_3), \\
& Y \downarrow \hat{X}_0, \hat{X}_0 \downarrow \hat{X}_1 \downarrow \hat{X}_2 \downarrow \hat{X}_3, \hat{X}_0 \downarrow \hat{X}'_1 \downarrow \hat{X}'_2 \downarrow \hat{X}'_3, \hat{X}_0 \downarrow \hat{X}''_1 \downarrow \hat{X}''_2 \downarrow \hat{X}''_3, \\
& X_1 \sim \hat{X}_1, X'_1 \sim \hat{X}'_1, \dots, X''_3 \sim \hat{X}''_3, \\
& X_3 \downarrow Z_1, a_1(Z_1), X'_3 \downarrow Z_2, a_2(Z_2), X''_3 \downarrow Z_3, a_3(Z_3), \\
& \hat{X}_3 \downarrow \hat{Z}_1, b_1(\hat{Z}_1), \hat{X}'_3 \downarrow \hat{Z}_2, b_2(\hat{Z}_2), \hat{X}''_3 \downarrow \hat{Z}_3, b_3(\hat{Z}_3).
\end{aligned}$$

And there are another two rules to deal with the case when the least common ancestor is labeled with $(L, 1)$ or $(R, 1)$

$$\begin{aligned}
G_{\mathcal{Q}}(X) \leftarrow & \top(X), X \downarrow Y, Y \downarrow X_0, X \downarrow \hat{Y}, X_0 \sim \hat{Y}, \\
& X_0 \downarrow X_1, X_1 \downarrow X_2 \downarrow X_3, X_1 \downarrow X'_2 \downarrow X'_3, X_1 \downarrow X''_2 \downarrow X''_3, \\
& (L, 2)(X_2), (L, 3)(X_3), \\
& (L, 2)(X'_2), (R, 3)(X'_3), \\
& (R, 2)(X''_2), (L, 3)(X''_3), \\
& Y \downarrow \hat{X}_0 \downarrow \hat{X}_1, \hat{X}_1 \downarrow \hat{X}_2 \downarrow \hat{X}_3, \hat{X}_1 \downarrow \hat{X}'_2 \downarrow \hat{X}'_3, \hat{X}_1 \downarrow \hat{X}''_2 \downarrow \hat{X}''_3, \\
& X_1 \sim \hat{X}_1, X_2 \sim \hat{X}_2, \dots, X_3 \sim \hat{X}_3, \\
& X_3 \downarrow Z_1, a_1(Z_1), X'_3 \downarrow Z_2, a_2(Z_2), X''_3 \downarrow Z_3, a_3(Z_3), \\
& \hat{X}_3 \downarrow \hat{Z}_1, b_1(\hat{Z}_1), \hat{X}'_3 \downarrow \hat{Z}_2, b_2(\hat{Z}_2), \hat{X}''_3 \downarrow \hat{Z}_3, b_3(\hat{Z}_3) \\
G_{\mathcal{Q}}(X) \leftarrow & \top(X), X \downarrow Y, Y \downarrow X_0, X \downarrow \hat{Y}, X_0 \sim \hat{Y}, \\
& X_0 \downarrow X_1, X_1 \downarrow X_2 \downarrow X_3, X_1 \downarrow X'_2 \downarrow X'_3, X_1 \downarrow X''_2 \downarrow X''_3, \\
& (L, 2)(X_2), (R, 3)(X_3), \\
& (R, 2)(X'_2), (L, 3)(X'_3), \\
& (R, 2)(X''_2), (R, 3)(X''_3), \\
& Y \downarrow \hat{X}_0 \downarrow \hat{X}_1, \hat{X}_1 \downarrow \hat{X}_2 \downarrow \hat{X}_3, \hat{X}_1 \downarrow \hat{X}'_2 \downarrow \hat{X}'_3, \hat{X}_1 \downarrow \hat{X}''_2 \downarrow \hat{X}''_3, \\
& X_1 \sim \hat{X}_1, X_2 \sim \hat{X}_2, \dots, X_3 \sim \hat{X}_3, \\
& X_3 \downarrow Z_1, a_1(Z_1), X'_3 \downarrow Z_2, a_2(Z_2), X''_3 \downarrow Z_3, a_3(Z_3), \\
& \hat{X}_3 \downarrow \hat{Z}_1, b_1(\hat{Z}_1), \hat{X}'_3 \downarrow \hat{Z}_2, b_2(\hat{Z}_2), \hat{X}''_3 \downarrow \hat{Z}_3, b_3(\hat{Z}_3).
\end{aligned}$$

□

B.6 Proof of Lemma 7

Take programs $\mathcal{P} \in \text{Datalog}(\downarrow)$ and $\mathcal{Q} \in \text{UCQ}(\downarrow)$. For every query φ in \mathcal{Q} consider the pattern π_φ . Each of these patterns corresponds to a tree t_φ which is unique up to renaming of labels that are not explicitly mentioned by \mathcal{Q} . Additionally, t_φ has one marked node X corresponding to the head node of φ . It remains to check if $\mathcal{P}(X)$ holds for each of these trees. It is well known that the combined complexity of monadic programs is NPTIME-complete. For each t_φ it suffices to guess the proof tree and verify the correctness of the guess.

C Boundedness

Proof (of Proposition 9). The 'only if' part is obvious. For the 'if' part, suppose that a datalog program \mathcal{P} is equivalent to a union of conjunctive queries \mathcal{Q} . For every rule φ of \mathcal{Q} consider a pattern π_φ . With each of these patterns we associate a set of trees: the possible homomorphic images of π_φ . Up to renaming of the

labels which are not explicitly mentioned by \mathcal{Q} there are finitely many such trees (this is because φ is connected and does not use the relation \downarrow_+). We evaluate the program \mathcal{P} on each of these trees and take n to be the biggest number of applications of the rules in \mathcal{P} that we need. Now let t be any tree. We will show that $\mathcal{P}(t) = \mathcal{P}^n(t)$. To this end, consider a node X of t such that $\mathcal{P}(X)$. Since the programs \mathcal{P} and \mathcal{Q} are equivalent, $\mathcal{Q}(X)$ also holds. This means that for some CQ φ of \mathcal{Q} there is a witnessing homomorphism h from π_φ to t . Thus, we need at most n applications of the rules in \mathcal{P} to derive $\mathcal{P}(X)$, because $h(\pi_\varphi)$ is a fragment of t . \square

C.1 Undecidability of the boundedness problem in general

Proof (of Theorem 11). We will reduce the following problem: given a Turing machine M , are there arbitrary long runs of M that start from an empty tape and end in the halting state (denoted HALT). This problem is undecidable, because for a machine M , for every transition of M that goes from state q seeing symbol a on tape to HALT state, we add another transition that stays in the state q after reading a and does not change the position of M 's head. Thus, if M had a run that halted, modified M has arbitrary long halting runs.

Let M be a Turing machine. We can assume without loss of generality that M has one tape, semi-infinite to the right. We will construct two programs, P and Q . Program P will find the encoding of the run of M on an empty input in the tree and Q will detect errors in the encoding. The Q program will be equivalent to a union of an UCQ. Moreover, we will ensure that for every correct run of M , there is only one corresponding encoding. Our program P_M will be an alternative of P and Q :

$$\begin{aligned} P_M(X) &: \neg P(X) \\ P_M(X) &: \neg Q(X) \end{aligned}$$

If a tree contains an error in the encoding, P_M will hold for every node of the tree in just 3 steps of the computation, because Q will be equivalent to an UCQ. The constructed program will be *not bounded* if and only if M has arbitrary long halting runs.

The run of M will be encoded as a word describing consecutive configurations. Configurations will be separated by $\#$ symbols. The beginning of the encoding will be a START symbol and the end will be denoted by END. Each position on the tape will be encoded by 4 consecutive nodes, $R - N - C - T$ where R will denote row number, N the number of the next row, C the column number and T the encoded tape symbol. s will be marked with 0 or 1 denoting if the head of M is in this position. Because we consider trees, the encoding will be placed in the tree from some node upwards to the root. This way, the program will have only one path on which it can match. Otherwise (that is, going downwards in the tree) the correctness of the encoding cannot be guaranteed.

For each transition τ of M , there will be a set of rules verifying that the two consecutive encoded configurations of M are consistent with τ . Single rules will

verify that the contents of the tape are copied/changed correctly between the configurations. To ensure that, the rule will look at each 3 consecutive positions. For each triple of tape symbols, there will be rule that matches 3 positions encoding those tape symbols. A rule $P_{\tau, a_1, a_2, a_3}^i(X)$ is true in X if 3 positions described directly above X contain symbols a_1, a_2, a_3 and the symbol in the next configuration in the same position as a_2 is also consistent with τ . If the head of tape, this symbol should just be copied, but if head of M is in the position with a_1, a_2 or a_3 the symbol can change between configurations. The $i = 1$ if the head of the tape was already seen in this configuration, 0 otherwise. For example, for a position where the head has not been seen in this configuration and there is no head in the inspected positions:

$$\begin{aligned}
P_{\tau, (0, s_1), (0, s_2), (0, s_3)}^0(R_1) : - \\
& T_3 \downarrow C_3 \downarrow N_3 \downarrow R_3 \downarrow T_2 \downarrow C_2 \downarrow N_2 \downarrow R_2 \downarrow T_1 \downarrow C_1 \downarrow N_1 \downarrow R_1 \\
& T_5 \downarrow C_5 \downarrow N_5 \downarrow R_5 \downarrow T_4 \downarrow C_4 \downarrow N_4 \downarrow R_4 \downarrow_+ T_3 \\
& (0, s_1)(T_1), (0, s_2)(T_2), (0, s_3)(T_3), (0, s_2)(T_5) \\
& R_4 \sim R_5, R_4 \sim N_1, C_5 \sim C_2, C_4 \sim C_1 \\
& N_1 \sim N_2 \sim N_3, R_1 \sim R_2 \sim R_3, R_4 \sim R_5 \\
& P_{\tau, (0, s_2), (0, s_3), (i, s_4)}^0(R_1)
\end{aligned}$$

There will be such rule for any possible tape symbol (i, s_4) . A quadruple R_i, N_i, C_i, T_i of variables describes one position of the tape, in the configuration R_i , with next configuration N_i and in column C_i . The symbol stored in this position is T_i . Additionally, there will be rules for changing rows, that checks two last positions before the $\#$ and ensures that the next row is either the same length as the previous one or one position longer (that is, has 4 more nodes), depending on the movement of the head. There will be also rules for the final row of the encoding (that is after reaching halting state), P_{fin} . P_{fin} will just go to the last $\#$, and P will be true in the root of the tree (with END label) if P_{fin} is matched in the last $\#$:

The program Q is given below, where Q_{err} is an alternative of all possible errors in the encoding.

$$Q(X) : -Y \downarrow_+ X, Y \downarrow_+ Z, Q_{err}(Z) \quad (1)$$

$$Q(X) : -Q_{err}(X) \quad (2)$$

$$Q(X) : -X \downarrow_+ Y, Q_{err}(Y) \quad (3)$$

Note the necessity of this triple alternative as \downarrow_+ is a proper descendant relation, that is $X \downarrow_+ X$ does not hold. This way, Q holds in every node of the tree if Q_{err} is found anywhere. The possible errors are

1. $\#$ or tape symbol appearing on the wrong position, for example detecting symbol $(0, s)$ used as a column number

$$Q_{err}(X) : -\#(X), X_3 \downarrow X_2 \downarrow X_1 \downarrow X, (0, s)(X_3)$$

$$Q_{err}(X) : -\#(X), X_3 \downarrow X_2 \downarrow X_1 \downarrow_+ Y \downarrow X, \sim (Y, X_1), (0, s)(X_3)$$

Similarly such rules can be constructed for next row, row and # used a tape symbol.

2. two consecutive # symbols, detected by $Q_{err}(X) : -\#(X), X \downarrow Y, \#(Y)$.
3. any node appears above the END, detected by $Q_{err}(X) : -\text{END}(X), Y \downarrow X$
4. any node appears below the START, detected by $Q_{err}(X) : -\text{START}(X), X \downarrow Y$
5. row number used in two different rows, detected by

$$Q_{err}(X) : -\#(X), Y_1 \downarrow X, Y_2 \downarrow_+ Y_1, \#(Y_2), Z \downarrow Y_2, Z \sim Y_1$$

6. the same column number twice in one row, detected by

$$Q_{err}(X) : -\#(X), Z_3 \downarrow Z_2 \downarrow Z_1 \downarrow_+ Y_3 \downarrow Y_2 \downarrow Y_1 \downarrow X \\ Y_1 \sim Z_1, Y_3 \sim Z_3$$

The last program works only if every row has distinct row number, which is ensured by previous rule.

It is easy to see that P_M is matched in every node of any tree that contains one of described errors, and in the root node of those databases that contain correct encoding of a halting run of M . Moreover, the computation of P_M in those databases takes number of steps linearly proportional to the length of the encoding. Therefore, P_M is unbounded if and only if M has the arbitrary long halting run property. \square

C.2 Boundedness on words and ranked trees

Proof (of Lemma 14). One implication is immediate. If \mathcal{P} is bounded then it is equivalent to a union of conjunctive queries \mathcal{Q} . The queries are connected so we can take n to be the size of the biggest query in \mathcal{Q} .

For the other implication, let us assume that \mathcal{P} satisfies the condition:

- there exists $n > 0$ such that for every word w and position X if $X \in \mathcal{P}(w)$ then $X \in \mathcal{P}(v)$, where v is the n -neighbourhood of X in w

with $n = n_0$. We will construct a union of conjunctive queries \mathcal{Q} equivalent to \mathcal{P} . Recall that $\Sigma_{\mathcal{P}}$ denotes the set of labels that appear in the rules of program \mathcal{P} . Let us consider all words of length smaller or equal $2n_0 + 1$ and treat them as structures over the signature $\{\downarrow, \sim\} \cup \Sigma_{\mathcal{P}}$. These words have finitely many equality types. For each word v that satisfies \mathcal{P} we add to \mathcal{Q} a query which defines the equality type of v . It remains to show that \mathcal{P} and \mathcal{Q} are equivalent. The containment of \mathcal{Q} in \mathcal{P} is straightforward from the construction of \mathcal{Q} . Take a word w and position X such that $X \in \mathcal{P}(w)$. Then $X \in \mathcal{P}(v)$, where v is the n -neighbourhood of X in w . Since v is a word of length at most $2n_0 + 1$ it follows that $X \in \mathcal{Q}(v)$, and hence $X \in \mathcal{Q}(w)$. \square

We now move to the case of trees. First let us state the lemma equivalent to Lemma 14 for ranked trees. For a tree t , the n -neighbourhood of a node X is a subtree of t consisting of all nodes that are in distance at most n from X .

Lemma 26. *Let \mathcal{P} be a $\text{Datalog}(\downarrow)$ program over ranked trees. Then the following conditions are equivalent:*

1. \mathcal{P} is bounded,
2. there exists $n > 0$ such that for every tree t and node X if $X \in \mathcal{P}(t)$ then $X \in \mathcal{P}(t')$, where t' is the n -neighbourhood of X in t .

Proof. The proof is analogous to the proof of Lemma 14. Let k be the rank of the considered trees. To show the implication from 2 to 1 it is enough to notice that for given n there are finitely many equality types (with respect to \mathcal{P}) of trees of height at most $2n + 1$ (and thus, finitely many of equality types of n -neighbourhoods). The equality type of each such n -neighbourhood is definable by a CQ, and a UCQ equivalent to \mathcal{P} is a union of those CQ's that are contained in \mathcal{P} . \square

In the case of trees we define an n -witness for \mathcal{P} to be a tree t such that there exists a node X in t for which $X \in \mathcal{P}(t)$ but $X \notin \mathcal{P}(t')$, where t' is the n -neighbourhood of X in t . A witness is a tree that is an n -witness for any $n > 0$.

Corollary 27. *A $\text{Datalog}(\downarrow)$ program \mathcal{P} over ranked trees is unbounded iff there exist n -witnesses for arbitrarily big $n > 0$.*

We can now give the proof of Theorem 13. We restate it first.

Theorem. The boundedness problem for $\text{Datalog}(\downarrow)$ over ranked trees is in 2-EXPTIME.

Proof. To prove Theorem 13 we first show that boundedness can be verified over ranked trees over a finite alphabet.

Lemma 28. *Let \mathcal{P} be a $\text{Datalog}(\downarrow)$ program. Then \mathcal{P} is bounded over ranked data trees with rank R over Σ iff \mathcal{P} is bounded over ranked trees with the same rank over a finite alphabet $\Sigma_0 \subseteq \Sigma$. The alphabet Σ_0 contains $\Sigma_{\mathcal{P}}$ and $|\Sigma_0 \setminus \Sigma_{\mathcal{P}}| \leq R^{|\mathcal{P}|}$.*

Proof. This proof is a slight modification of a proof from [17]. If \mathcal{P} is bounded over Σ then it is clearly bounded over any finite subset of Σ . Suppose that \mathcal{P} is bounded over Σ_0 but not bounded over Σ . Over Σ_0 , \mathcal{P} is therefore equivalent to a UCQ Q built of a finite number of proof words of \mathcal{P} . Let t be a tree over Σ and X a node in t s.t. $X \in \mathcal{P}(t)$ but $X \notin Q(t)$. We will show that t can be relabeled into a tree t' over Σ_0 in a way preserving any label comparison done by the rules of \mathcal{P} . Then, as Q is a union of proof words of \mathcal{P} , it must also hold that $X \in Q(t)$ iff $X \in Q(t')$, which is a contradiction since \mathcal{P} is not equivalent to Q over ranked trees over Σ .

Let n be the size of the largest rule in \mathcal{P} . Let $B \subseteq \Sigma \setminus \Sigma_{\mathcal{P}}$ be a set of size $R^{|\mathcal{P}|}$. We set $\Sigma_0 = B \cup \Sigma_{\mathcal{P}}$. We will describe a procedure that traverses the tree t in a top-down fashion, level by level, and changes the labels to elements of B . This way the set of processed nodes consists of i full levels starting from the root, and some nodes from the level $i + 1$.

Let v be a node on level $i + 1$ – the next one 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 that the label of v is a . If $a \in B \cup \Sigma_{\mathcal{P}}$, we can finish processing v . Assume that $a \notin B \cup \Sigma_{\mathcal{P}}$. Pick a label $b \in B$ that does not appear in the processed descendants of u , nor in u itself. We can always find such a label b because the number of processed descendants of u (including u itself) is bounded by $\sum_{i=0}^{n-1} R^i = \frac{R^n - 1}{R - 1} < R^n \leq R^{|\mathcal{P}|}$, and so is the number of labels from B used in these nodes. Let $c \in \Sigma \setminus (B \cup \Sigma_{\mathcal{P}})$ be a fresh label. We now replace all appearances of b with c , but only in the unprocessed descendants of the node u . Observe that these nodes are separated from the 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 the 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 in \mathcal{P} (because they use only the short axis and are connected). After all nodes are processed, all labels in t' are from $B \cup \Sigma_{\mathcal{P}}$.

Let Σ_0 be the finite alphabet from the previous Lemma. Now we can construct an automaton $W_{\mathcal{P}}$, recognizing the set of witnesses for \mathcal{P} . From Lemma 21 we get a two-way alternating tree automaton $\mathcal{B}_{\mathcal{P}}$ which works over $\Sigma_0 \times \{0, 1\}$, and accepts the set of trees that have only one node labeled with $(a, 1)$ for $a \in \Sigma_0$, and the goal predicate of \mathcal{P} is satisfied in this node. The size of this automaton is exponential in $|\mathcal{P}|$. Let $\mathcal{A}_{\mathcal{P}}$ be the bottom-up automaton recognizing $L(\mathcal{B}_{\mathcal{P}})$ obtained via Proposition 18. Let $\mathcal{N}_{\mathcal{P}}$ be an automaton obtained by taking a product of the bottom-up automaton recognizing the complement of $L(\mathcal{B}_{\mathcal{P}})$ (again obtained via Proposition 18) and the automaton checking that there is only one node in the tree with label $(a, 1)$ for some $a \in \Sigma_0$. Then $\mathcal{N}_{\mathcal{P}}$ accepts all trees over Σ_0 for which \mathcal{P} does not hold in the marked node. The size of both $\mathcal{A}_{\mathcal{P}}$ and $\mathcal{N}_{\mathcal{P}}$ is double exponential in $|\mathcal{P}|$.

With those two automata, the construction of $W_{\mathcal{P}}$ is easy. The set of states of $W_{\mathcal{P}}$ is

$$Q(\mathcal{A}_{\mathcal{P}}) \times (\{\epsilon, \text{OK}\} \cup Q(\mathcal{N}_{\mathcal{P}}))$$

where $Q(A)$ denotes the set of states of the automaton A . Let t be a tree over $\Sigma_0 \times \{0, 1\}$ and let X denote the marked node. The automaton $W_{\mathcal{P}}$ starts in the state (q_I, ϵ) , where q_I is the initial state of $\mathcal{A}_{\mathcal{P}}$. Then $W_{\mathcal{P}}$ simulates $\mathcal{A}_{\mathcal{P}}$ on t . In any node of a tree, the automaton $W_{\mathcal{P}}$ can guess that here begins the neighbourhood of X in which \mathcal{P} does not hold. Then $W_{\mathcal{P}}$ changes the second component of its state from ϵ to the initial state of $\mathcal{N}_{\mathcal{P}}$ and simulates $\mathcal{N}_{\mathcal{P}}$ on the guessed neighbourhood, verifying that indeed \mathcal{P} does not hold in it. If $W_{\mathcal{P}}$ has reached an accepting state of $\mathcal{N}_{\mathcal{P}}$, it can guess that this node is the root of the neighbourhood and change the state to OK in the second component. Accepting states of $W_{\mathcal{P}}$ are states (q, OK) where q is any accepting state of $\mathcal{A}_{\mathcal{P}}$.

Similarly to the word case, if there exists a witness of size linear in the size of the automaton $W_{\mathcal{P}}$, then there exist arbitrarily big witnesses.

Lemma 29. *Let N be the number of states of the automaton $W_{\mathcal{P}}$. If there exists a $(2N + 2)$ -witness for \mathcal{P} , then there exist n -witnesses for arbitrary large n . The existence of $(2N + 2)$ -witness can be decided in time polynomial in N .*

Proof. We use a very similar pumping argument as in the word case. This time, however, to obtain arbitrarily big witnesses we need to be able to pump every path of the neighbourhood in which \mathcal{P} is not satisfied.

Suppose that there exists a $(2N + 2)$ -witness and let X be the marked node. Then on every path of length $2N + 2$ from X downwards, some state of $W_{\mathcal{P}}$ must repeat, so we can pump the context between those nodes. Notice that some paths may be shorter, because the $(2N + 2)$ -witness may contain a leaf of the tree – we don't need to pump those paths. On the path from X upwards of length $N + 1$ again some states of $W_{\mathcal{P}}$ repeat, and we can pump the context between the occurrences of the same state. This time, however, we need also to extend the paths that start on the pumped fragment and go downwards, but do not return to X . Every such path is of length at least $N + 1$ (that is why we need the $2N + 2$ size of the neighbourhood), so we can pump each of them (except for those that are shorter because they end with a leaf of the tree).

To verify the existence of a $(2N + 2)$ -witness we modify the automaton $W_{\mathcal{P}}$ by adding two counters from 0 to $2N + 2$. When the automaton guesses the beginning of a neighbourhood of X in a non-leaf node Y it starts counting the length of the shortest path until the least common ancestor of Y and X is reached. The automaton in a node calculates the length of the shortest path as $1 +$ the minimum of the values of the counters calculated for its children (if the value of the counter is $2N + 2$, adding 1 does not change its value). When a neighbourhood of X begins in a leaf of the tree, the length of this path does not need to be $2N + 2$, so the automaton sets the counter to $2N + 2$ (that is – sufficient length). The second counter is used only for the nodes on the path above X and counts the length of the path for X to this node (for any other node in the guessed neighbourhood, value of this counter is 0).

It is not difficult to see that using those two counters we can come up with an acceptance condition such that the modified automaton has an accepting run iff there exists a $(2N + 2)$ -witness for \mathcal{P} . Since emptiness can be decided in time linear in the size of the automaton, we get the claim. \square

Since the size of $W_{\mathcal{P}}$ is double exponential in $|\mathcal{P}|$, we get a 2-EXPTIME procedure for deciding boundedness of \mathcal{P} . \square