

Query Containment for Highly Expressive Datalog Fragments

Pierre Bourhis, Markus Krötzsch, Sebastian Rudolph

▶ **To cite this version:**

Pierre Bourhis, Markus Krötzsch, Sebastian Rudolph. Query Containment for Highly Expressive Datalog Fragments. 2014. hal-01098974

HAL Id: hal-01098974

https://inria.hal.science/hal-01098974

Preprint submitted on 30 Dec 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Query Containment for Highly Expressive Datalog Fragments

Pierre Bourhis
CNRS LIFL University of Lille 1
& INRIA Lille Nord Europe
Lille, FR

Markus Krötzsch
Fakultät Informatik
Technische Universität
Dresden, DE

Sebastian Rudolph
Fakultät Informatik
Technische Universität
Dresden, DE

ABSTRACT

The containment problem of Datalog queries is well known to be undecidable. There are, however, several Datalog fragments for which containment is known to be decidable, most notably monadic Datalog and several "regular" query languages on graphs. Monadically Defined Queries (MQs) have been introduced recently as a joint generalization of these query languages.

In this paper, we study a wide range of Datalog fragments with decidable query containment and determine exact complexity results for this problem. We generalize MQs to (Frontier-)Guarded Queries (GQs), and show that the containment problem is 3ExpTime-complete in either case, even if we allow arbitrary Datalog in the sub-query. If we focus on graph query languages, i.e., fragments of linear Datalog, then this complexity is reduced to 2ExpSpace. We also consider nested queries, which gain further expressivity by using predicates that are defined by inner queries. We show that nesting leads to an exponentially increasing hierarchy for the complexity of query containment, both in the linear and in the general case. Our results settle open problems for (nested) MQs, and they paint a comprehensive picture of the state of the art in Datalog query containment.

1. INTRODUCTION

Query languages and their mutual relationships are a central topic in database research and a continued focus of intensive study. It has long been known that first-order logic expressions over the database relations (represented by *extensional database predicates*, EDBs) lack the expressive power needed in many scenarios. Higher-order query languages have thus been introduced, which allow for the recursive definition of new predicates (so called *intensional database predicates*, IDBs). Most notably, Datalog has been widely studied as a very expressive query language with tractable query answering (w.r.t. the size of the database).

On the other hand, Datalog has been shown to be too expressive a language for certain tasks which are of crucial importance in database management. In particular, the *query containment problem* that, given two queries Q_1 and Q_2, asks if every answer to Q_1 is an answer to Q_2 in every possible database, is undecidable for full Datalog [21]. However, checking query containment is an essential task facilitating query optimization, information integration and exchange, as well as database integrity checking. It comes handy for utilizing databases with materialized views and, as part of an offline preprocessing technique, and it may help accelerating online query answering.

This motivates the question for Datalog fragments that are still expressive enough to satisfy their purposes but exhibit decidable query containment. Moreover, once decidability is established, the precise complexity of deciding containment provides further insights. The pursuit of these issues has led to a productive and well-established line of research in database theory, which has already produced numerous results for a variety of Datalog fragments.

Non-recursive Datalog and unions of conjunctive queries. A non-recursive Datalog program does not have any (direct or indirect) recursion and it is equivalent to a union of conjunctive queries (UCQ) (and thus expressible in first-order logic). The problem of containment of a Datalog program (in the following referred to as Dlog) in a union of conjunctive queries is 2ExpTime-complete [14]. Due to the succintness of non-recursive Datalog compared to UCQs, the problem of containment of Dlog in non-recursive Datalog is 3ExpTime-complete [14]. Some restrictions for decreasing the complexity of these problems have been considered. Containment of *linear* Datalog programs (LinDlog), i.e., one where rule bodies contain at most one IDB in a UCQ, is ExpSpace-complete; complexity further decreases to PSpace when the linear Datalog program is monadic (LinMDlog, see below) [13, 14].

The techniques to prove the upper bounds in these results are based on the reduction to the problem of containment of tree automata for the general case, and to the containment of word automata in the linear case.

Monadic Datalog. A monadic Datalog (MDlog) program is a program containing only unary intensional predicates. The problem of containment for MDlog is 2ExpTime complete. The upper bound is well known since the 80's [15], while the lower bound has been established only recently [6]. Finally, the containment of Dlog in a monadic MDlog

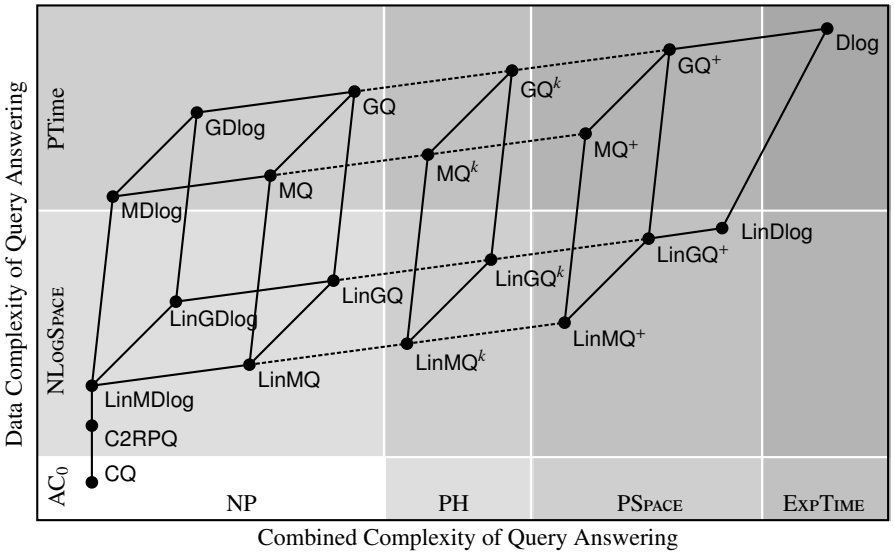


Figure 1: Query languages and complexities; languages higher up in the graph are more expressive

is also decidable. It is a straightforward application of Theorem 5.5 of [16].[1] So far, however, tight bounds have not been known for this result.

Guarded Datalog. Guarded Datalog (GDlog) allows the use of intensional predicates with unrestricted arities, however for each rule, the variables of the head should appear in a single extensional atom appearing in the body of the rule. While this notion of (frontier-)guarded rules is known for a while [8, 3], the first use of GDlog as a query language seems to be only recent [4]. GDlog is a proper extension of MDlog, since monadic rules can always be rewritten into guarded rules [4]. It is know that query containment for GDlog is 2ExpTime-complete, a result based on the decidability of the satisfiability of the guarded negation fixed point logic [5].

Navigational Queries. Conjunctive two-way regular path queries (C2RPQs) generalize conjunctive queries (CQs) by regular expressions over binary predicates [18, 9]. Variants of this type of queries are used, e.g., by the XPath query language for querying semi-structured XML data. Recent versions of the SPARQL 1.1 query language for RDF also support some of regular expressions that can be evaluated under a similar semantics. Intuitively, C2RPQ is a conjunct of atoms of the form xLy where L is a two-way regular expression. A pair of nodes $\langle n_1, n_2 \rangle$ is a valuation of the pair $\langle x, y \rangle$ if and only if there exists a path between n_1 and n_2 matching L. The containment of queries in this language was shown to be ExpSpace-complete [18, 10, 2, 17]. The containment of Dlog in C2RPQ is 2ExpTime-complete [11].

Monadically Defined Queries. More recently, Monadically Defined Queries (MQs) and their nested version (MQ$^+$s) have been introduced [19] as a proper generalization of MDlog which also captures (unions of) C2RPQs. At the same time, they are conveniently expressible both in Dlog and monadic second-order logic. Yet, as opposed to these two, MQs and MQ$^+$s have been shown to have a decidable containment problem, but no tight bounds were known so far.

In spite of these continued efforts, the complexity of query containment is still unclear for many well-known Datalog fragments, especially for the most expressive ones. In this paper, we thus study a variety of known and new query languages in more detail. Figure 1 gives an overview of all Datalog fragments we consider, together with their respective query-answering complexities.

We provide a detailed complexity analysis of the mutual containment between queries of the aforementioned (and some new) formalisms. This analysis is fine-grained in the sense that—in the case of query formalisms that allow for nesting—precise complexities depending on the nesting depth are presented. Moreover, we consider the case where the used rules are restricted to linear Datalog.

- We introduce *guarded queries* (GQs) and their nested versions (GQ$^+$s), Datalog fragments that properly generalize MQs and MQ$^+$s, respectively, while featuring the same data and combined complexities for query answering. On the other hand, already unnested GQs subsume GDlog. We also consider the restrictions of all these queries to the linear Datalog case and observe that this drops data complexities to NLogSpace whereas it does not affect combined complexities.

- By means of sophisticated automata-based techniques involving iterated transformations on alternating two-way automata, we show a generic upper bound stating that containment of Dlog in nested guarded queries

[1] We thank Michael Benedikt for this observation.

of depth k (GQ^k) can be decided in $(k+2)$ExpTime. Additionally we show that going down to GDlog on the containment's right-hand side allows deciding it in 2ExpTime.

- Inductively defining alternating Turing machine simulations on tapes of $(k+1)$-exponential size, we provide a matching generic lower bound by showing that containment of MDlog in MQ^k is $(k+2)$ExpTime-hard. Together with the upper bound, this provides precise complexities for all cases, where the left-hand side of the containment is any fragment between MDlog and Dlog (cf. Fig. 1) and the right-hand side is any of MQ, GQ, MQ^k, GQ^k, MQ^+, GQ^+. In particular, this solves the respective open questions from [19]: MQ containment is 3ExpTime-complete and MQ^+ containment is NonElementary.
- We next investigate the situation in case only linear rules are allowed in the definition of the Datalog fragment used on the left hand side of the containment problem (this distinction generally makes no difference for the right-hand side). We find that in most of these cases, the complexities mentioned above drop to $(k+1)$ExpSpace.

In summary, our results settle open problems for (nested) MQs, and they paint a comprehensive and detailed picture of the state of the art in Datalog query containment.

2. PRELIMINARIES

We consider a standard language of first-order predicate logic, based on an infinite set $\mathbf{C}$ of *constant symbols*, an infinite set $\mathbf{P}$ of *predicate symbols*, and an infinite set $\mathbf{V}$ of first-order *variables*. Each predicate $p \in \mathbf{P}$ is associated with a natural number $\mathsf{ar}(p)$ called the *arity* of p. The list of predicates and constants forms the language's *signature* $\mathscr{S} = \langle \mathbf{P}, \mathbf{C} \rangle$. We generally assume $\mathscr{S} = \langle \mathbf{P}, \mathbf{C} \rangle$ to be fixed, and only refer to it explicitly if needed.

Formulae, Rules, and Queries. A *term* is a variable $x \in \mathbf{V}$ or a constant $c \in \mathbf{C}$. We use symbols s, t to denote terms, x, y, z, v, w to denote variables, a, b, c to denote constants. Expressions like $\boldsymbol{t}$, $\boldsymbol{x}$, $\boldsymbol{c}$ denote finite lists of such entities. We use the standard predicate logic definitions of *atom* and *formula*, using symbols φ, ψ for the latter.

Datalog queries are defined over an extended signature with additional predicate symbols, called *IDB predicates*; all other predicates are called *EDB predicates*. A *Datalog rule* is a formula of the form $\forall \boldsymbol{x}, \boldsymbol{y}.\varphi[\boldsymbol{x}, \boldsymbol{y}] \rightarrow \psi[\boldsymbol{x}]$ where φ and ψ are conjunctions of atoms, called the *body* and *head* of the rule, respectively, and where ψ only contains IDB predicates. We usually omit universal quantifiers when writing rules. Sets of Datalog rules will be denoted by symbols $\mathbb{P}, \mathbb{R}, \mathbb{S}$. A set of Datalog rules $\mathbb{P}$ is

- *monadic* if all IDB predicates are of arity one;
- *frontier-guarded* if the body of every rule contains an atom $p(\boldsymbol{t})$ such that p is an EDB predicate and $\boldsymbol{t}$ contains all variables that occur in the rule's head;
- *linear* if every rule contains at most one IDB predicate in its body.

A *conjunctive query* (CQ) is a formula $Q[\boldsymbol{x}] = \exists \boldsymbol{y}.\psi[\boldsymbol{x}, \boldsymbol{y}]$ where $\psi[\boldsymbol{x}, \boldsymbol{y}]$ is a conjunction of atoms; a *union of conjunctive queries* (UCQ) is a disjunction of such formulae. A *Datalog query* $\langle \mathbb{P}, Q \rangle$ consists of a set of Datalog rules $\mathbb{P}$ and a conjunctive query Q over IDB or EDB predicates (Q could be expressed as a rule in Datalog, but not in all restrictions of Datalog we consider). We write Dlog for the language of Datalog queries. A monadic Datalog query is one where $\mathbb{P}$ is monadic, and similarly for other restrictions. We use the query languages MDlog (monadic), GDlog (frontier-guarded), LinDlog (linear), and LinMDlog (linear, monadic).

Databases and Semantics. We use the standard semantics of first-order logic (FOL). A *database instance* $\mathcal{I}$ consists of a set $\Delta^{\mathcal{I}}$ called *domain* and a function $\cdot^{\mathcal{I}}$ that maps constants c to domain elements $c^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ and predicate symbols p to relations $p^{\mathcal{I}} \subseteq (\Delta^{\mathcal{I}})^{\mathsf{ar}(p)}$, where $p^{\mathcal{I}}$ is the *extension* of p.

Given a database instance $\mathcal{I}$ and a formula $\varphi[\boldsymbol{x}]$ with free variables $\boldsymbol{x} = \langle x_1, \ldots, x_m \rangle$, the *extension* of $\varphi[\boldsymbol{x}]$ is the subset of $(\Delta^{\mathcal{I}})^m$ containing all those tuples $\langle \delta_1, \ldots, \delta_m \rangle$ for which $\mathcal{I}, \{x_i \mapsto \delta_i \mid 1 \leq i \leq m\} \models \varphi[\boldsymbol{x}]$. We denote this by $\langle \delta_1, \ldots, \delta_m \rangle \in \varphi^{\mathcal{I}}$ or by $\mathcal{I} \models \varphi(\delta_1, \ldots, \delta_m)$; a similar notation is used for all other types of query languages. Two formulae $\varphi[\boldsymbol{x}]$ and $\psi[\boldsymbol{x}]$ are called *equivalent* if their extensions coincide for every database instance $\mathcal{I}$.

The set of answers of a UCQ $Q[\boldsymbol{x}]$ over $\mathcal{I}$ is its extension. The set of answers of a Datalog query $\langle \mathbb{P}, Q \rangle$ over $\mathcal{I}$ is the intersection of the extensions of Q over all extended database instances $\mathcal{I}'$ that interpret IDB predicates in such a way that all rules of $\mathbb{P}$ are satisfied. Datalog [1] can also be defined as the least fixpoint on the inflationary evaluation of Q on I.

Note that we do not require database instances to have a finite domain, since all of our results are valid in either case. This is due to the fact that every entailment of a Datalog program has a finite witness, and that all of our query languages are positive, i.e., that their answers are preserved under homomorphisms of database instances.

3. GUARDED QUERIES

Monadically defined queries have been introduced in [19] as a generalization of monadic Datalog (MDlog) and conjunctive two-way regular path queries (C2RPQs) for which query containment is still decidable.[2] The underlying idea of this approach is that candidate query answers are checked by evaluating a monadic Datalog program, i.e., in contrast to the usual evaluation of Datalog queries, we start with a "guessed" answer that is the input to a Datalog program.

[2]The queries were called MODEQ in [19]; we shorten this to MQ.

To implement this, the candidate answer is represented by special constants λ that the Datalog program can refer to. This mechanism was called *flag & check*, since the special constants act as flags to indicate the answer that should be checked.

Example 1. *A query that computes the transitive closure over a relation p can be defined as follows.*

$$p(\lambda_1, y) \to \mathtt{U}(y)$$
$$\mathtt{U}(y) \wedge p(y, z) \to \mathtt{U}(z)$$
$$\mathtt{U}(\lambda_2) \to \mathsf{hit}$$

One defines the answer of the query to contain all pairs $\langle \delta_1, \delta_2 \rangle$ for which the rules entail hit *when interpreting λ_1 as δ_1 and λ_2 as δ_2.*

The approach used monadic Datalog for its close relationship to monadic second-order logic, which was the basis for showing decidability of query containment. In this work, however, we develop new techniques for showing the decidability (and exact complexity) of this problem directly. It is therefore suggestive to consider other types of Datalog programs to implement the "check" part. The following definition therefore introduces the general technique for arbitrary Datalog programs, and defines interesting fragments by imposing further restrictions.

Definition 1. *Consider a signature $\mathscr{S}$. An FCP ("flag & check program") of arity m is a set of Datalog rules $\mathbb{P}$ with $k \geq 0$ IDB predicates $\mathtt{U}_1, \ldots, \mathtt{U}_k$, that may use the additional constant symbols $\lambda_1, \ldots, \lambda_m \notin \mathscr{S}$ and an additional nullary predicate symbol* hit*. An FCQ ("flag & check query") P is of the form $\exists \boldsymbol{y}.\mathbb{P}(\boldsymbol{z})$, where $\mathbb{P}$ is an FCP of arity $|\boldsymbol{z}|$ and all variables in $\boldsymbol{y}$ occur in $\boldsymbol{z}$. The variables $\boldsymbol{x}$ that occur in $\boldsymbol{z}$ but not in $\boldsymbol{y}$ are the* free variables *of P.*

Let $\mathcal{I}$ be a database instance over $\mathscr{S}$. The extension $\mathbb{P}^{\mathcal{I}}$ *of $\mathbb{P}$ is the set of all tuples $\langle \delta_1, \ldots, \delta_m \rangle \in (\Delta^{\mathcal{I}})^m$ such that every database instance $\mathcal{I}'$ that extends $\mathcal{I}$ to the signature of $\mathbb{P}$ and that satisfies $\langle \lambda_1^{\mathcal{I}'}, \ldots, \lambda_m^{\mathcal{I}'} \rangle = \langle \delta_1, \ldots, \delta_m \rangle$ also entails* hit*. The semantics of FCQs is defined in the obvious way based on the extension of FCPs.*

A GQ *is an FCQ $\exists \boldsymbol{y}.\mathbb{P}(\boldsymbol{z})$ such that $\mathbb{P}$ is frontier-guarded. Similarly, we define* MQ *(monadic),* LinMQ *(linear, monadic), and* LinGQ *(linear, frontier-guarded) queries.*

In contrast to [19], we do not define monadic queries as conjunctive queries of FCPs, but we merely allow existential quantification to project some of the FCP variables. Proposition 2 below shows that this does not reduce expressiveness.

We generally consider monadic Datalog as a special case of frontier-guarded Datalog. Monadic Datalog rules do not have to be frontier-guarded. A direct way to obtain a suitable guard is to assume that there is a unary domain predicate that contains all (relevant) elements of the domain of the database instance. However, it already suffices to require *safety* of Datalog rules, i.e., that the variable in the head of a rule must also occur in the body. Then every element that is inferred to belong to an IDB relation must also occur in some EDB relation. We can therefore add single EDB guard atoms to each rule in all possible ways without modifying the semantics. This is a polynomial operation, since all variables in the guards are fresh, other than the single head variable that we want to guard. We therefore find, in particular, the GQ captures the expressiveness of MQ. The converse is not true, as the following example illustrates.

Example 2. *The following 4-ary* LinGQ *generalizes Example 1 by checking for the existence of two parallel p-chains of arbitrary length, where each pair of elements along the chains is connected by a relation q, like the steps of a ladder.*

$$q(\lambda_1, \lambda_2) \to \mathtt{U}_q(\lambda_1, \lambda_2)$$
$$\mathtt{U}_q(x, y) \wedge p(x, x') \wedge p(y, y'), q(x', y') \to \mathtt{U}_q(x', y')$$
$$\mathtt{U}_q(\lambda_3, \lambda_4) \to \mathsf{hit}$$

One might assume that the following MQ *is equivalent:*

$$q(\lambda_1, \lambda_2) \to \mathtt{U}_1(\lambda_1)$$
$$q(\lambda_1, \lambda_2) \to \mathtt{U}_2(\lambda_2)$$
$$\mathtt{U}_1(x) \wedge \mathtt{U}_2(y) \wedge p(x, x') \wedge p(y, y'), q(x', y') \to \mathtt{U}_1(x')$$
$$\mathtt{U}_1(x) \wedge \mathtt{U}_2(y) \wedge p(x, x') \wedge p(y, y'), q(x', y') \to \mathtt{U}_2(y')$$
$$\mathtt{U}_1(\lambda_3) \wedge \mathtt{U}_2(\lambda_4) \to \mathsf{hit}$$

However, the latter query also matches structures that are not ladders. For example, the following database yields the answer $\langle a, b, c, d \rangle$, although there is no corresponding ladder structure: $\{q(a, b), p(a, c), p(b, e), q(c, e), p(a, e'), p(b, d), q(e', d)\}$. One can extend the MQ *to avoid this case, but any such fix is "local" in the sense that a sufficiently large ladder-like structure can trick the query.*

It has been shown that monadically defined queries can be expressed both in Datalog and in monadic second-order logic [19]. While we lose the connection to monadic second-order logic with GQs, the expressibility in Datalog remains. The encoding is based on the intuition that the choice of the candidate answers for $\boldsymbol{\lambda}$ "contextualizes" the inferences of the Datalog program. To express this without special constants, we can store this context information in predicates of suitably increased arity.

Example 3. *The 4-ary* LinGQ *of Example 2 can be expressed with the following Datalog query. For brevity, let $\boldsymbol{y}$ be the variable list $\langle y_1, y_2, y_3, y_4 \rangle$, which provides the context for the IDB facts we derive.*

$$q(y_1, y_2) \to \mathtt{U}_q^+(y_1, y_2, \boldsymbol{y})$$
$$\mathtt{U}_q(x, y, \boldsymbol{y}) \wedge p(x, x') \wedge p(y, y'), q(x', y') \to \mathtt{U}_q^+(x', y', \boldsymbol{y})$$
$$\mathtt{U}_q(y_3, y_4, \boldsymbol{y}) \to \mathsf{goal}(\boldsymbol{y})$$

This result is obtained by a straightforward extension of the translation algorithm for MQ*s [19], which may not produce*

the most concise representation. Also note that the first rule in this program is not safe, since y_3 and y_4 occur in the head but not in the body. According to the semantics we defined, such variables can be bound to any element in the active domain of the given database instance (i.e., they behave as if bound by a unary domain *predicate).*

This observation justifies that we consider MQs, GQs, etc. as Datalog fragments. It is worth noting that the translation does not change the number of IDB predicates in the body of rules, and thus preserves linearity. The relation to (linear) Datalog also yields some complexity results for query answering; we will discuss these at the end of the next section, after introducing nested variants our query languages.

4. NESTED QUERIES

Every query language gives rise to a nested language, where we allow nested queries to be used as if they were predicates. Sometimes, this does not lead to a new query language (like for CQ and Dlog), but often it affects complexities and/or expressiveness. It has been shown that both are increased when moving from MQs to their nested variants [19]. We will see that nesting also has strong effects on the complexity of query containment.

Definition 2. *We define k-nested FCPs inductively. A* 1*-nested FCP is an FCP. A* $k+1$*-nested FCP is an FCP that may use k-nested FCPs of arity m instead of predicate symbols of arity m in rule bodies. The semantics of nested FCPs is immediate based on the extension of FCPs. A k-nested FCQ P is of the form* $\exists \boldsymbol{y}.\mathbb{P}(\boldsymbol{z})$*, where* $\mathbb{P}$ *is a k-nested FCP of arity* $|\boldsymbol{z}|$ *and all variables in* $\boldsymbol{y}$ *occur in* $\boldsymbol{z}$*.*

A k-nested GQ *query is a k-nested frontier-guarded FCQ. For the definition of* frontier-guarded*, we still require EDB predicates in guards: subqueries cannot be guards. The language of k-nested* GQ *queries is denoted* GQ^k*; the language of arbitrarily nested* GQ *queries is denoted* GQ^+*. Similarly, we define languages* MQ^k *and* MQ^+ *(monadic),* LinMQ^k *and* LinMQ^+ *(linear, monadic), and* LinGQ^k *and* LinGQ^+ *(linear, frontier-guarded).*

Note that nested queries can use the same additional symbols (predicates and constants); this does not lead to any semantic interactions, however, as the interpretation of the special symbols is "private" to each query. To simplify notation, we assume that distinct (sub)queries always contain distinct special symbols. The relationships of the query languages we introduced here are summarized in Figure 1, where upwards links denote increased expressiveness. An interesting observation that is represented in this figure is that linear Datalog is closed under nesting:

Theorem 1. $\mathsf{LinDlog} = \mathsf{LinDlog}^+$.

Another kind of nesting that does not add expressiveness is the nesting of FCQs in UCQs. Indeed, it turns out that (nested) FCQs can internalize arbitrary conjunctions and disjunctions of FCQs (of the same nesting level). This even holds when restricting to linear rules.

Proposition 2. *Let P be a positive query, i.e., a Boolean expression of disjunctions and conjunctions, of* LinMQ^k *queries with* $k \geq 1$*. Then there is a* LinMQ^k *query* P' *of size polynomial in P that is equivalent to P. Analogous results hold when replacing* LinMQ^k *by* MQ^k*,* GQ^k*, or* LinMQ^k *queries.*

Query answering for MQs has been shown to be NP-complete (combined complexity) and P-complete (data complexity). For MQ^+, the combined complexity increases to PSpace while the data complexity remains the same. These results can be extended to frontier-guarded queries. We also note the query complexity for frontier-guarded Datalog, for which we are not aware of any published result.

Theorem 3. *The combined complexity of evaluating* GQ *queries over a database instance is* NP*-complete. The same holds for* GDlog *queries. The combined complexity of evaluating* GQ^+ *queries is* PSpace*-complete. The data complexity is* P*-complete for* GDlog*,* GQ*, and* GQ^+.

The lower bounds in the previous case are immediate from know results for monadically defined queries. In particular, the hardness proof for nested MQs also shows that queries of a particular fixed nesting level can encode the validity problem for quantified boolean formulae with a certain number of quantifier alternations; this explains why we show the combined complexity of MQ^k to be in the Polynomial Hierarchy in Figure 1. A modification of this hardness proof from [19] allows us to obtain the same results for the combined complexities in the linear cases; matching upper bounds follow from Theorem 3.

Theorem 4. *The combined complexity of evaluating* LinMQ *queries over a database instance is* NP*-complete. The same holds for* LinGDlog *and* LinGQ*. The combined complexity of evaluating* LinMQ^+ *queries is* PSpace*-complete. The same holds for* LinGQ^+.

The data complexity is NLogSpace*-complete for all of these query languages.*

5. DECIDING QUERY CONTAINMENT WITH AUTOMATA

We first recall a general technique of reducing query containment to the containment problem for (tree) automata [14], which we build our proofs on. An introduction to tree automata is included in the appendix.

A common way to describe the answers of a Dlog query $P = \langle \mathbb{P}, p \rangle$ is to consider its *expansion trees*. Intuitively speaking, the goal atom $p(\boldsymbol{x})$ can be rewritten by applying rules of $\mathbb{P}$ in a backward-chaining manner until all IDB predicates have been eliminated, resulting in a CQ. The answers

of P coincide with the (infinite) union of answers to the CQs obtained in this fashion. The rewriting itself gives rise to a tree structure, where each node is labeled by the instance of the rule that was used in the rewriting, and the leaves are instances of rules that contain only EDB predicates in their body. The set of all expansion trees provides a regular description of P that we exploit to decide containment.

To formalize this approach, we describe the set of all expansion trees as a tree language, i.e., as a set of trees with node labels from a finite alphabet. The number of possible labels of nodes in expansion trees is unbounded, since rules are instantiated using fresh variables. To obtain a finite alphabet of labels, one limits the number of variables and thus the overall number of possible rule instantiations [14].

Definition 3. *Given a* Dlog *query* $P = \langle \mathbb{P}, p \rangle$, $\mathcal{R}_\mathbb{P}$ *is the set of all instantiations of rules of* $\mathbb{P}$ *using only the variables* $\mathcal{V}_\mathbb{P} = \{v_1, \ldots, v_n\}$, *where* n *is twice the maximal number of variables occurring in any rule of* $\mathbb{P}$.

A proof tree *for* P *is a tree with labels from* $\mathcal{R}_\mathbb{P}$, *such that (a) the root is labeled by a rule with* p *as its head predicate; (b) if a node is labeled by a rule* ρ *with an IDB atom* B *in its body, then it has a child node that is labeled by* ρ' *with head atom* B. *The label of a node* e *is denoted* $\pi(e)$.

Consider two nodes e_1 *and* e_2 *in a proof tree with lowest common ancestor* e. *Two occurrences of a variable* v *in* $\pi(e_1)$ *and* $\pi(e_2)$ *are* connected *if* v *occurs in the head of* $\pi(f)$ *for all nodes* f *on the shortest path between* e_1 *and* e_2, *with the possible exception of* e.

A proof tree encodes an expansion tree where we replace every set of mutually connected variable occurrences by a fresh variable. Conversely, every expansion tree is represented by a proof tree that replaces fresh body variables by variables that do not occur in the head; this is always possible since proof trees can use twice as many variables as any rule of $\mathbb{P}$. The set of proof trees is a regular tree language that can be described by an automaton.

Proposition 5 (Proposition 5.9 [14]). *For a* Dlog *query* $P = \langle \mathbb{P}, p \rangle$, *there is a tree automaton* $\mathcal{A}_P$ *of size exponential in* P *that accepts exactly the set of all proof trees of* P.

In order to use $\mathcal{A}_P$ to decide containment of P in another query P', we construct an automaton $\mathcal{A}_{P \sqsubseteq P'}$ that accepts all proof trees of P that are "matched" by P'. Indeed, every proof tree induces a *witness*, i.e., a minimal matching database instance, and one can check whether or not P' can produce the same query answer on this instance. If this is the case for all proof trees of P, then containment is shown.

6. DECIDING GUARDED QUERY CONTAINMENT

Our first result provides the upper bound for deciding containment of GQ queries. In fact, the result extends to arbitrary Dlog queries on the left-hand side.

Theorem 6. *Containment of* Dlog *queries in* GQ *queries can be decided in* 3ExpTime.

To prove this, we need to construct the tree automaton $\mathcal{A}_{P \sqsubseteq P'}$ for an arbitrary GQ P'. As a first step, we construct an alternating 2-way tree automaton $\mathcal{A}^+_{P \sqsubseteq P'}$ that accepts the proof trees that we would like $\mathcal{A}_{P \sqsubseteq P'}$ to accept, but with nodes additionally being annotated with information about the choice of λ values to guide the verification.

We first construct automata to verify the match of a single, non-recursive rule that may refer to λ constants. The rule does not have to be monadic or frontier-guarded. Our construction is inspired by a similar construction for CQs by Chaudhuri and Vardi [14], with the main difference that the answer variables in our case are not taken from the root of the tree but rather from one arbitrary node that is marked accordingly.

To define this formally, we introduce trees with additional annotations besides their node labels. Clearly, such trees can be viewed as regular labelled trees by considering annotations to be components of one label; our approach, however, leads to a more readable presentation.

Definition 4. *Consider a Datalog program* $\mathbb{P}$, *a rule* $\rho = \varphi \to p(\boldsymbol{x})$, *and* $n \geq 0$ *special constants* $\boldsymbol{\lambda} = \lambda_1, \ldots, \lambda_n$. *The proof-tree variables* $\mathcal{V}_\mathbb{P}$ *used in* $\mathcal{R}_\mathbb{P}$ *are as in Definition 3.*

A proof tree for $\mathbb{P}$ *is* $\boldsymbol{\lambda}$-annotated *if every node has an additional* $\boldsymbol{\lambda}$-label *that is a partial mapping* $\{\lambda_1, \ldots, \lambda_n\} \to \mathcal{V}_\mathbb{P}$, *such that: every special constant* λ_i *occurs in at least one* $\boldsymbol{\lambda}$*-label, and whenever a constant* λ_i *occurs in two* $\boldsymbol{\lambda}$*-labels, it is mapped to the same variable and both variable occurrences are connected.*

A proof tree for $\mathbb{P}$ *is* p-annotated *if exactly one node has an additional* p-label *of the form* $p(\boldsymbol{v})$, *where* $\boldsymbol{v}$ *is a list of variables from* $\mathcal{V}_\mathbb{P}$.

A matching tree T *for* ρ *and* $\mathbb{P}$ *is a* $\boldsymbol{\lambda}$*-annotated and* p*-annotated proof tree for* $\mathbb{P}$ *for which there is a mapping* $\nu : \mathsf{Var}(\rho) \cup \{\lambda_1, \ldots, \lambda_n\} \to \mathcal{V}_\mathbb{P}$ *such that*

(a) $\nu(p(\boldsymbol{x})) = p(\boldsymbol{v})$;

(b) for every atom α *of* φ, *there is a node* e_α *in* T *such that the rule instance that* e_α *is labeled with contains the EDB atom* $\nu(\alpha)$ *in its body;*

(c) if λ_i *occurs in* α, *then the* $\boldsymbol{\lambda}$*-label maps* λ_i *to the occurrence of* $\nu(\lambda_i)$ *in* e_α;

(d) if $\alpha, \alpha' \in \varphi$ *share a variable* x, *then the occurrences of* $\nu(x)$ *in* e_α *and* $e_{\alpha'}$ *are connected.*

Proposition 7. *There is an automaton* $\mathcal{A}_{P,\rho}$ *that accepts exactly the annotated matching trees for* ρ *and* $\mathbb{P}$, *and which is exponential in the size of* ρ *and* $\mathbb{P}$.

We want to use the automata $\mathcal{A}_{P,\rho}$ to verify the entailment of a single rule within a Datalog derivation. We would like an automaton to check whether a whole derivation is possible. Unfortunately, we cannot check these derivations using

automata of the form $\mathcal{A}_{P,\rho}$, which each need to be run on a p-annotated tree which has the unique entailment of the rule marked. The length of a derivation is unbounded, and we would not be able to distinguish an unbounded amount of p-markers. To overcome this problem, we create a modified automaton $\mathcal{A}^+_{P,\rho,\boldsymbol{v}}$ that simulates the behavior of $\mathcal{A}_{P,\rho}$ on a tree with annotation $p(\boldsymbol{v})$. For $\mathcal{A}^+_{P,\rho,\boldsymbol{v}}$ to know which node the annotation $p(\boldsymbol{v})$ refers to, it has to be started at this node. This is a non-standard notion of run, where we do not start at the root of the tree. Moreover, starting in the middle of the tree makes it necessary to consider both nodes below and above the current position, and $\mathcal{A}^+_{P,\rho,\boldsymbol{v}}$ therefore needs to be an *alternating 2-way tree automaton*.

Proposition 8. *There is an alternating 2-way tree automaton $\mathcal{A}^+_{P,\rho,\boldsymbol{v}}$ that is polynomial in the size of $\mathcal{A}_{P,\rho}$ such that, whenever $\mathcal{A}_{P,\rho}$ accepts a matching tree T that has the p-annotation $p(\boldsymbol{v})$ on node e, then $\mathcal{A}^+_{P,\rho,\boldsymbol{v}}$ has an accepting run that starts from the corresponding node e' on the tree T' that is obtained by removing the p-annotation from T.*

Using the automata $\mathcal{A}^+_{P,\rho,\boldsymbol{v}}$, we can now obtain the claimed alternating 2-way automaton $\mathcal{A}^+_{P\sqsubseteq P'}$ for a GQ P'. Intuitively speaking, $\mathcal{A}^+_{P\sqsubseteq P'}$ concatenates the automata $\mathcal{A}^+_{P,\rho,\boldsymbol{v}}$ using alternation: whenever a derivation requires a (recursive) IDB atom, a suitable process $\mathcal{A}^+_{P,\rho,\boldsymbol{v}}$ is initiated, starting from a node in the middle of the tree. The construction relies on guardedness, which ensures that we can always find a suitable start node (corresponding to the node that was p-annotated earlier), by finding a suitable guard EDB atom in the tree.

Proposition 9. *For a* Dlog *query P and a* GQ *query P' with special constants $\boldsymbol{\lambda}$, there is an alternating 2-way automaton $\mathcal{A}^+_{P\sqsubseteq P'}$ of exponential size that accepts the $\boldsymbol{\lambda}$-annotated proof trees of P that encode expansion trees with $\boldsymbol{\lambda}$ assignments for which P' has a match.*

We are now ready to prove Theorem 6. The automaton $\mathcal{A}^+_{P\sqsubseteq P'}$ allows us to check the answers of P' on a proof tree that is $\boldsymbol{\lambda}$-annotated to assign values for answer constants. We can transform this alternating 2-way automaton into a tree automaton $\mathcal{A}'_{P\sqsubseteq P'}$ that is exponentially larger, i.e., doubly exponential in the size of the input. To remove the need for $\boldsymbol{\lambda}$-labels, we modify the automaton $\mathcal{A}'_{P\sqsubseteq P'}$ so that it can only perform a transition from its start state if it finds that the constants in $\boldsymbol{\lambda}$ are assigned to the answer variables of P in the root. Finally, we obtain $\mathcal{A}_{P\sqsubseteq P'}$ by projecting to the alphabet $\mathcal{R}_{\mathbb{P}}$ without $\boldsymbol{\lambda}$-annotations; this is again possible in polynomial effort. The containment problem $P \sqsubseteq P'$ is equivalent by deciding the containment of $\mathcal{A}_P$ in $\mathcal{A}_{P\sqsubseteq P'}$, which is possible in exponential time w.r.t. to the size of the automata. Since $\mathcal{A}_P$ is exponential and $\mathcal{A}_{P\sqsubseteq P'}$ is double exponential, we obtain the claimed triple exponential bound.

Our proof of Theorem 6 can be used to obtain another interesting result for the case of frontier-guarded Datalog. If P is a GDlog query, which does not use any special constants λ, then the $\boldsymbol{\lambda}$-annotations are not relevant and $\mathcal{A}^+_{P\sqsubseteq P'}$ can be constructed as an alternating 2-way automaton on proof trees. For this, we merely need to modify the construction in Proposition 9 to start in start states of automata for rules that entail the goal predicate of P' with the expected binding of variables to answer variables of P. We can then omit the projection step, which required us to convert $\mathcal{A}^+_{P\sqsubseteq P'}$ into a tree automaton earlier. Instead, we can construct from $\mathcal{A}^+_{P\sqsubseteq P'}$ a complement tree automaton $\bar{\mathcal{A}}_{P\sqsubseteq P'}$ that is only exponentially larger than $\mathcal{A}^+_{P\sqsubseteq P'}$, i.e., doubly exponential overall [15][Theorem A.1]. Containment can then be checked by checking the non-emptiness of $\mathcal{A}_P \cap \bar{\mathcal{A}}_{P\sqsubseteq P'}$, which is possible in polynomial time, leading to a 2ExpTime algorithm overall.

Theorem 10. *Containment of* Dlog *queries in* GDlog *queries can be decided in* 2ExpTime.

This generalizes an earlier result of Cosmadakis et al. for monadic Datalog [15] using an alternative, direct proof.

Finally, we can lift our results to the case of nested queries. Using Proposition 2, we can make the simplifying assumption that rules with some nested query in their body contain only one nested query and a guard atom as the only other atom. Thus all rules with nested queries have the form $g(\boldsymbol{s}) \wedge Q(\boldsymbol{t}) \rightarrow p(\boldsymbol{u})$, where g is an EDB predicate, Q is a nested query, and the variables $\boldsymbol{u}$ occur in $\boldsymbol{s}$.

In Proposition 8, we constructed alternating 2-way automata $\mathcal{A}^+_{P,\rho,\boldsymbol{v}}$ that can check the entailment of a particular atom $p(\boldsymbol{v})$ starting from a node within the tree. Analogously, we now construct automata $\mathcal{A}^+_{P,Q,\theta}$ that check that the nested query Q matches partially, where θ is a substitution that interprets query variables in terms of proof-tree variables on the current node of the tree. Only the variables that occur in $g(\boldsymbol{s})$ and $Q(\boldsymbol{t})$ are mapped by θ; the remaining variables can be interpreted arbitrarily, possibly in distant parts of the proof tree.

To construct $\mathcal{A}^+_{P,Q,\theta}$, we use the alternating 2-way automaton $\mathcal{A}^+_{P\sqsubseteq Q}$, constructed in Proposition 9 (assuming, for a start, that Q is not nested). This automaton is extended to an alternating 2-way automaton $\mathcal{A}^+_{P,Q}$ that accepts trees with a unique annotation of the form $\langle Q, \theta\rangle$, for which we check that it is consistent with the $\boldsymbol{\lambda}$-annotation (i.e., for each query variable x mapped by θ, the corresponding constant λ is assigned to $\theta(x)$ at the node that is annotated with $\langle Q, \theta\rangle$). We then obtain a (top-down) tree automaton $\mathcal{A}_{P,Q}$ by transforming $\mathcal{A}^+_{P,Q}$ into a tree automaton (exponential), and projecting away the $\boldsymbol{\lambda}$-annotations (polynomial). The automaton $\mathcal{A}_{P,Q}$ is analogous to the tree automaton $\mathcal{A}_{P,\rho}$ of Proposition 7. Using the same transformation as in Proposition 8, we obtain an alternating 2-way automaton $\mathcal{A}^+_{P,Q,\theta}$ for each θ.

The automaton $\mathcal{A}^+_{P\sqsubseteq P'}$ for a nested query P' is constructed as in Proposition 9, but using the automata $\mathcal{A}^+_{P,Q,\theta}$ instead of automata $\mathcal{A}^+_{P,\rho,\boldsymbol{v}}$ to check the entailment of a subquery Q. The size of $\mathcal{A}^+_{P\sqsubseteq P'}$ is increased by one exponential since the size of $\mathcal{A}^+_{P,Q,\theta}$ is exponentially increased when projecting out

λ-labels for Q. Applying this construction inductively, we obtain the following result.

THEOREM 11. *Containment of* Dlog *queries in* GQ^k *queries can be decided in* $(k+2)$EXPTIME.

7. SIMULATING ALTERNATING TURING MACHINES

To show the hardness of query containment problems, we generally provide direct encodings of Alternating Turing Machines (ATMs) with a fixed space bound [12]. To simplify this encoding, we assume without loss of generality that every universal ATM configuration leads to exactly two successor configurations. The following definition defines ATM encodings formally. Rather than requiring concrete structures to encode ATMs, we abstract the encoding by means of queries that find suitable structures in a database instance; this allows us to apply the same definition for increasingly complex encodings. The following definition is illustrated in Figure 2.

DEFINITION 5. *Consider an ATM* $\mathcal{M} = \langle Q, \Sigma, \Delta, q_s, q_e\rangle$ *and queries* $\mathsf{FirstConf}[x,y]$, $\mathsf{NextConf}_\delta[x,y]$ *for all* $\delta \in \Delta$, $\mathsf{LastConf}[x]$, $\mathsf{State}_q[x]$ *for all* $q \in Q$, $\mathsf{Head}[x,y]$, $\mathsf{ConfCell}[x,y]$, $\mathsf{FirstCell}[x,y]$, $\mathsf{NextCell}[x,y]$, $\mathsf{LastCell}[x]$, *and* $\mathsf{Symbol}[x,y]$. *To refer to tape symbols, we consider constants* c_σ *for all* $\sigma \in \Sigma$, *and to refer to positions of the head, we use constants h (here), l (left), and r (right).*

With respect to these queries, an element $c \in \mathsf{dom}(\mathcal{I})$ *in a database instance* $\mathcal{I}$ encodes an $\mathcal{M}$ quasi-configuration of size s *if* $\mathcal{I}$ *contains a structure*

$$\begin{array}{l}\mathsf{State}_q(c), \mathsf{FirstCell}(c,d_1),\\ \mathsf{ConfCell}(c,d_1), \mathsf{Symbol}(d_1,c_{\sigma_1}), \mathsf{Head}(d_1,p_1), \mathsf{NextCell}(d_1,d_2),\\ \mathsf{ConfCell}(c,d_2), \mathsf{Symbol}(d_2,c_{\sigma_2}), \mathsf{Head}(d_2,p_2), \ldots, \mathsf{NextCell}(d_{s-1},d_s),\\ \mathsf{ConfCell}(c,d_s), \mathsf{Symbol}(d_s,c_{\sigma_s}), \mathsf{Head}(d_s,p_s), \mathsf{LastCell}(d_s),\end{array}$$

where $q \in Q$, $\sigma_i \in \Sigma$, *and* $p_i \in \{h,l,r\}$. *We say that* c encodes an $\mathcal{M}$ configuration of size s *if, in addition, the sequence* $(p_i)_{i=1}^s$ *has the form* $l,\ldots,l,h,r,\ldots,r$ *with zero or more occurrences of r and l, respectively.*

An element c *in* $\mathcal{I}$ encodes a (quasi-)configuration tree of $\mathcal{M}$ in space s *if*

- $\mathcal{I} \models \mathsf{FirstConf}(c,d_1)$ *for some* d_1,
- d_1 *is the root of a tree with edges defined by* $\mathsf{NextConf}_\delta$,
- *every node in this tree encodes an* $\mathcal{M}$ *(quasi-) configuration of size s,*
- *if there is a transition* $\mathcal{I} \models \mathsf{NextConf}_{\delta_1}(e,e_1)$, *where* $\delta_1 = \langle q,\sigma,q',\sigma',d\rangle$ *and q is a universal state, then there is also a transition* $\mathcal{I} \models \mathsf{NextConf}_{\delta_2}(e,e_2)$ *with* $\delta_1 \neq \delta_2$,
- *if e is a leaf node, then* $\mathcal{I} \models \mathsf{LastConf}(e)$.

If the tree is an accepting run, then c encodes an accepting run (of $\mathcal{M}$ *in space s).*

A same-cell query *is a query* $\mathsf{SameCell}[x,y]$ *such that, if* $c_1, c_2 \in \mathsf{dom}(\mathcal{I})$ *encode two quasi-configurations, and* $d_1, d_2 \in \mathsf{dom}(\mathcal{I})$ *represent the same tape cell in the encodings* c_1 *and* c_2, *respectively, then* $\langle d_1,d_2\rangle \in \mathsf{SameCell}^{\mathcal{I}}$.

Two queries $P_1[x]$ *and* $P_2[x]$ containment-encode *accepting runs of* $\mathcal{M}$ *in space s if, for every database instance* $\mathcal{I}$ *and element* $c \in P_1^{\mathcal{I}} \setminus P_2^{\mathcal{I}}$, *c encodes an accepting run of* $\mathcal{M}$ *in space s, and every accepting run of* $\mathcal{M}$ *in space s is encoded by some* $c \in P_1^{\mathcal{I}} \setminus P_2^{\mathcal{I}}$ *for some* $\mathcal{I}$.

Note that elements c may encode more than one configuration (or configuration tree). This is not a problem in our arguments.

The conditions that ensure that a quasi-configuration tree is an accepting run can be expressed by a query, based on the queries given in Definition 5. More specifically, one can construct a query that accepts all elements that encode a quasi-configuration sequence that is *not* a run. Together with a query that accepts only encodings of quasi-configurations tree, this allows us to containment-encode accepting runs of an ATM. Only linear queries, possibly nested, will be needed to perform the required checks, even in the case of ATMs. To simplify the statements, we use LinMQ^0 as a synonym for UCQ.

LEMMA 12. *Consider an ATM* $\mathcal{M}$, *and queries as in Definition 5, including* $\mathsf{SameCell}[x,y]$, *that are* MQ^k *queries for some* $k \geq 0$. *There is a* MQ^k *query* $P[x]$, *polynomial in the size of* $\mathcal{M}$ *and the given queries, such that the following hold.*

- *For every accepting run of* $\mathcal{M}$ *in space s, there is some database instance* $\mathcal{I}$ *with some element c that encodes the run, such that* $c \notin P^{\mathcal{I}}$.
- *If an element c of* $\mathcal{I}$ *encodes a tree of quasi-configurations of* $\mathcal{M}$ *in space s, and if* $c \notin P^{\mathcal{I}}$, *then c encodes an accepting run of* $\mathcal{M}$ *in space s.*

Moreover, if all input queries are in LinMQ^k, *then so is P.*

The previous result allows us to focus on the encoding of quasi-configuration trees and the definition of queries as required in Definition 5. Indeed, the main challenge below will be to enforce a sufficiently large tape for which we can still find a correct same-cell query.

8. HARDNESS OF MONADIC QUERY CONTAINMENT

We can now prove our first major hardness result:

THEOREM 13. *Deciding containment of* MDlog *queries in* MQ^k *queries is hard for* $(k+2)$EXPTIME.

Note that the statement includes the 3EXPTIME-hardness for containment of MQs as a special case. To prove this result, we first construct an EXPSPACE ATM that we then use to construct tapes of double exponential size.

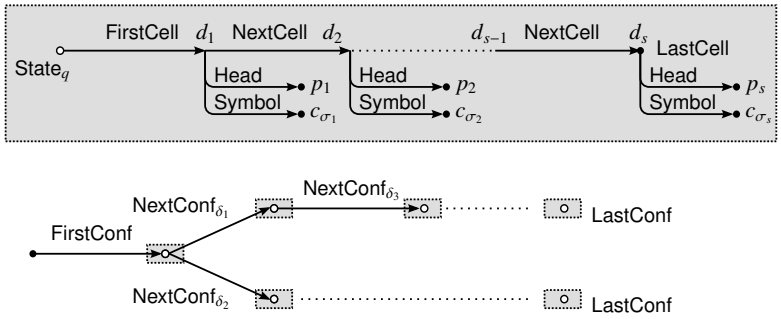


Figure 2: Illustration of the ATM encoding of Definition 5: shaded configurations (top) are used within the configuration tree (bottom); ConfCell queries are omitted for clarity

Lemma 14. *For any ATM $\mathcal{M}$, there is an* MDlog *query $P_1[x]$, a* LinMQ *$P_2[x]$, queries as in Definition 5 that are* LinMQ*s, and a same-cell query that is a* UCQ*, such that $P_1[x]$ and $P_2[x]$ containment-encode accepting runs of $\mathcal{M}$ in exponential space.*

Figure 3 illustrates the encoding that we use to prove Lemma 14. While it resembles the structure of Figure 2, the labels are now EDB predicates rather than (abstract) queries. The encoding of tapes attaches to each cell an ℓ-bit address (where bits are represented by constants 0 and 1). We can use these bits to count from 0 to 2^ℓ to construct tapes of this length. The query on the left-hand side can only enforce that there are cells with bit addresses, not that they actually count; even the exact length of the tape is unspecified. The query on the right-hand side of the containment then checks that consecutive cells (in all tapes that occur in the configuration tree) represent successor addresses, and that the first and last address is as expected.

Another difference from Figure 2 is that we now treat configurations as linear structures, with a beginning and an end. In our representation of the configuration tree, we next configuration therefore connects to the last cell of the previous configuration's tape, rather than its start. We do this to ensure that the encoding works well even when restricting to linear queries. Indeed, the only non-linear rules in P_1 are used to enforce multiple successor configurations for universal states of an ATM. For normal TMs, even P_1 is in LinMDlog. The rules of the P_1 are as follows:

$$\mathsf{firstConf}(x,y) \wedge \mathtt{U}_{conf}(y) \to \mathtt{U}_{goal}(x)$$

$$\mathsf{state}_q(x) \wedge \mathsf{firstCell}(x,y) \wedge \mathtt{U}_{bit_1}(y) \to \mathtt{U}_{conf}(x) \qquad \text{for } q \in Q$$

$$\mathsf{bit}_{i-1}(x,0) \wedge \mathtt{U}_{bit_i}(x) \to \mathtt{U}_{bit_{i-1}}(x) \qquad \text{for } i \in \{2,\dots,\ell\}$$

$$\mathsf{bit}_{i-1}(x,1) \wedge \mathtt{U}_{bit_i}(x) \to \mathtt{U}_{bit_{i-1}}(x) \qquad \text{for } i \in \{2,\dots,\ell\}$$

$$\mathsf{symbol}(x,c_\sigma) \wedge \mathtt{U}_{symbol}(x) \to \mathtt{U}_{bit_\ell}(x) \qquad \text{for } \sigma \in \Sigma$$

$$\mathsf{head}(x,p) \wedge \mathtt{U}_{head}(x) \to \mathtt{U}_{symbol}(x) \qquad \text{for } p \in \{h,r,l\}$$

$$\mathsf{nextCell}(x,y) \wedge \mathtt{U}_{bit_1}(y) \to \mathtt{U}_{head}(x)$$

$$\mathsf{nextConf}_\delta(x,y) \wedge \mathtt{U}_{conf}(y) \to \mathtt{U}_{head}(x) \qquad \text{for } \delta = \langle q,\sigma,q',\sigma',d\rangle \text{ with } q \in Q_\exists$$

$$\mathsf{nextConf}_{\delta_1}(x,y_1) \wedge \mathtt{U}_{conf}(y_1) \wedge \mathsf{nextConf}_{\delta_2}(x,y_2) \wedge \mathtt{U}_{conf}(y_2) \to \mathtt{U}_{head}(x) \qquad \text{for } \delta_1 = \langle q,\sigma,q',\sigma',d\rangle,\ q \in Q_\forall, \text{ and } \delta_1 \neq \delta_2$$

$$\mathsf{lastConf}(x) \to \mathtt{U}_{head}(x)$$

Note that we do not enforce any structure to define the query ConfCell; this query is implemented by a LinMQ that navigates over an arbitrary number of cells within one configuration. This is the main reason why we need LinMQs rather than UCQs here.

We now use the exponential space ATM of Lemma 14 to encode the tape of 2ExpSpace ATM. The following result shows, that one can always obtain an exponentially larger tape by nesting linear queries on the right-hand side.

Lemma 15. *Assume that there is some space bound s such that, for every DTM $\mathcal{M}$, there is a* MDlog *query $P_1[x]$ and an* MQ^{k+1} *query $P_2[x]$ with $k \geq 0$, such that $P_1[x]$ and $P_2[x]$ containment-encode accepting runs of $\mathcal{M}$ in s, where the queries required by Definition 5 are* MQ^{k+1} *queries. Moreover, assume that there is a suitable same-cell query that is in* MQ^k.

Then, for every ATM $\mathcal{M}'$, there is a MDlog *query $P'_1[x]$, an* MQ^{k+1} *$P'_2[x]$, and* MQ^{k+1} *queries as in Definition 5, such that $P'_1[x]$ and $P'_2[x]$ containment-encode an accepting run of $\mathcal{M}'$ in space $s' \geq 2^s$. Moreover, the size of the queries for this encoding is polynomial in the size of the queries for the original encoding.*

We show this result by using a deterministic space-s Turing machine $\mathcal{M}$ to count from 0 to 2^s, which takes a fixed number $s' > 2^s$ of steps. We then use the encodings of accepting runs of $\mathcal{M}$ as encodings for tapes of the ATM $\mathcal{M}'$, where every configuration of $\mathcal{M}$ becomes a cell of $\mathcal{M}'$. All tapes simulated in this way are of equal length s'. Some queries required by Definition 5 are easy to obtain: for example, the new query $\mathsf{NextCell}'[x,y]$ is the query $\mathsf{NextConf}[x,y]$ of the encoding of $\mathcal{M}$. The most difficult to express is the new same-cell query, for which we use the following MQ^{k+1}:

$$\mathsf{FirstCell}(\lambda_1,x) \to \mathtt{U}_1(x)$$

$$\mathtt{U}_1(x) \wedge \mathsf{NextCell}(x,x') \to \mathtt{U}_1(x')$$

$$\mathsf{State}_q(\lambda_1) \wedge \mathsf{FirstCell}(\lambda_1,x) \wedge \mathsf{Symbol}(x,z) \wedge \mathsf{Head}(x,v) \wedge \mathsf{State}_q(\lambda_2) \wedge \mathsf{FirstCell}(\lambda_2,y) \wedge \mathsf{Symbol}(y,z) \wedge \mathsf{Head}(y,v) \to \mathtt{U}_2(y) \qquad \text{for all } q \in Q$$

$$\mathtt{U}_1(x) \wedge \mathtt{U}_2(y) \wedge \mathsf{SameCell}(x,y) \wedge \mathsf{NextCell}(x,x') \wedge \mathsf{Symbol}(x',z) \wedge \mathsf{Head}(x',v) \wedge \mathsf{NextCell}(y,y') \wedge \mathsf{Symbol}(y',z) \wedge \mathsf{Head}(y',v) \to \mathtt{U}_2(y')$$

$$\mathtt{U}_2(y) \wedge \mathsf{LastCell}(y) \to \mathsf{hit}$$

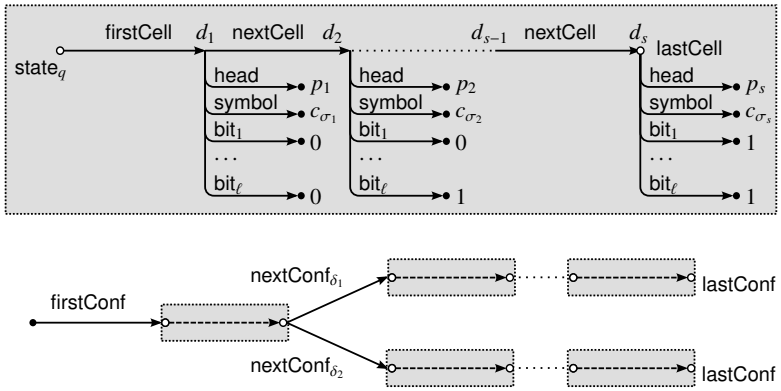


Figure 3: Illustration of the ATM encoding of Lemma 14: shaded configurations (top) are used within the configuration tree (bottom)

where FirstCell, Symbol, SameCell, and LastCell are the queries from the encoding of $\mathcal{M}$. The first two rules simply mark the tape starting at λ_1 with $\mathtt{U}_1$. The next two rules then compare the two (potentially very long) tapes from configurations of $\mathcal{M}$ to check if they contain exactly the same symbols at each position, and the last rule finishes. Since the tapes are not connected in any known way, we have to be careful to ensure never to loose the connection to either of the tapes, to avoid comparing random cells from other parts of the database. Indeed, the last two rules do not mention λ_1 or λ_2 at all. We need two IDB predicates to achieve this, which carefully mark the two tapes cell by cell.

Another important thing to note is that the query SameCell is only used exactly once in exactly one rule. Indeed, if we were using it twice, then the length of our queries would grow exponentially when applying the construction inductively. This is the reason why we encode symbols and head positions with constants, rather than using unary predicates like for states. In the latter case, we need many rules, one for each predicate, as can be seen in the third rule above. One could try to avoid the use of constants by more complex encodings that encode information using paths of different lengths as done by Björklund et al. [7]. However, some additional device is needed to ensure that database instances are sufficiently closely connected in this case, which may again require constants, IDBs of higher arity, or a greater nesting level of LinMQ queries to navigate larger distances.

With the previous results, Theorem 13 can be proved by an easy induction: for the base case $k = 1$ we apply Lemma 15 to the result of Lemma 14; for the induction step we use Lemma 15 again.

9. LINEAR DATALOG

Not only query answering, but also containment checking is often slightly simpler in fragments of linear Datalog. Intuitively, this is so because derivations can be represented as words rather than as trees. Thus, the automata theoretic techniques that we have used in Section 6 can be applied with automata on words where some operations are easier. In particular, containment of (nondeterministic) automata on words can be checked in polynomial space rather than in exponential time. This allows us to establish the following theorems, which reduce the 2ExpTime upper bound of Theorem 10 to ExpSpace and the $(k+2)$ExpTime upper bound of Theorem 11 to $(k+1)$ExpSpace.

Theorem 16. *Containment of* LinDlog *queries in* GDlog *queries can be decided in* ExpSpace.

Theorem 17. *Containment of* LinDlog *queries in* GQ^k *queries can be decided in* $(k+1)$ExpSpace.

Establishing matching lower bounds for the complexity turns out to be more difficult. In general, we loose the power of alternation, which explains the reduction in complexity. The general approach of encoding (non-alternating) Turing machines is the same as in Section 7, where Definition 5 is slightly simplified since we do not need to consider universal states, so that configuration trees turn into configuration sequences. Moreover, Lemma 12 applies to this case as well, since it only requires linear queries. Likewise, our general inductive step in Lemma 15 uses deterministic (non-alternating) TMs to construct exponentially long tapes. Moreover, it turns out that the construction of an initial exponential space TM in Lemma 14 leads to linear queries if the TM has no universal states.

Yet it is challenging to lift the exact encodings of Lemma 14 and Lemma 15. The same-cell query that we constructed in Lemma 15 for our inductive argument is nonlinear. As explained in Section 8, the use of two IDBs to mark both sequences of tape cells is essential there to ensure correctness. The main problem is that we must not loose connection to either of the sequences during our checks. As an alternative to using IDBs on both sequences, one could use the ConfCell query to ensure that the compared cells belong to the right configurations. This leads to the following same-cell query:

$$\mathsf{State}_q(\lambda_1) \wedge \mathsf{FirstCell}(\lambda_1, x) \wedge \mathsf{Symbol}(x, z) \wedge \mathsf{Head}(x, v) \wedge \mathsf{State}_q(\lambda_2) \wedge \mathsf{FirstCell}(\lambda_2, y) \wedge \mathsf{Symbol}(y, z) \wedge \mathsf{Head}(y, v) \rightarrow \mathtt{U}(y) \quad \text{for all } q \in Q$$

$$\mathtt{U}(y) \wedge \mathsf{ConfCell}(\lambda_1, x) \wedge \mathsf{SameCell}(x, y) \wedge \mathsf{NextCell}(x, x') \wedge \mathsf{Symbol}(x', z) \wedge \mathsf{Head}(x', v) \wedge \mathsf{NextCell}(y, y') \wedge \mathsf{Symbol}(y', z) \wedge \mathsf{Head}(y', v) \rightarrow \mathtt{U}(y')$$

$$\mathtt{U}(y) \wedge \mathsf{LastCell}(y) \rightarrow \mathsf{hit}$$

While this works in principle, it has the problem that the ConfCell query of Lemma 14 is a LinMQ, not a UCQ. Therefore, if we construct a same-cell query for the 2ExpSpace case, we obtain LinMQ^2 queries, which yields the following result:

Theorem 18. *Deciding containment of* LinMDlog *queries in* LinMQ^k *queries is hard for* kExpSpace.

In order to do better, one can try to express ConfCell as a UCQ. In general, this is not possible on the database instances that the left-hand query in Lemma 14 recognizes, since cells may have an exponential distance to their configuration while UCQs can only recognize local structures. To make ConfCell local, we can modify the left-hand query to ensure that every cell is linked directly to its configuration with a binary predicate inConf. Using binary IDB predicates, we can do this with the following set of frontier-guarded rules:

$$\mathsf{firstConf}(x, y) \wedge \mathtt{U}_{conf}(y) \rightarrow \mathtt{U}_{goal}(x)$$

$$\mathsf{state}_q(x) \wedge \mathsf{nextCell}(x, y) \wedge \mathsf{inConf}(y, x) \wedge \mathtt{U}_{bit_1}(y, x) \rightarrow \mathtt{U}_{conf}(x) \quad \text{for } q \in Q$$

$$\mathsf{bit}_{i-1}(x, 0) \wedge \mathtt{U}_{bit_i}(y, z) \wedge \mathsf{inConf}(x, z) \rightarrow \mathtt{U}_{bit_{i-1}}(x, z) \quad \text{for } i \in \{2, \ldots, \ell\}$$

$$\mathsf{bit}_{i-1}(x, 1) \wedge \mathtt{U}_{bit_i}(y, z) \wedge \mathsf{inConf}(x, z) \rightarrow \mathtt{U}_{bit_{i-1}}(x, z) \quad \text{for } i \in \{2, \ldots, \ell\}$$

$$\mathsf{symbol}(x, c_\sigma) \wedge \mathtt{U}_{symbol}(x, z) \wedge \mathsf{inConf}(x, z) \rightarrow \mathtt{U}_{bit_\ell}(x, z) \quad \text{for } \sigma \in \Sigma$$

$$\mathsf{head}(x, h) \wedge \mathtt{U}_{head}(x, z) \wedge \mathsf{inConf}(x, z) \rightarrow \mathtt{U}_{symbol}(x, z)$$

$$\mathsf{head}(x, l) \wedge \mathtt{U}_{head}(x, z) \wedge \mathsf{inConf}(x, z) \rightarrow \mathtt{U}_{symbol}(x, z)$$

$$\mathsf{head}(x, r) \wedge \mathtt{U}_{head}(x, z) \wedge \mathsf{inConf}(x, z) \rightarrow \mathtt{U}_{symbol}(x, z)$$

$$\mathsf{nextCell}(x, y) \wedge \mathtt{U}_{bit_1}(y, z) \wedge \mathsf{inConf}(x, z) \rightarrow \mathtt{U}_{head}(x, z)$$

$$\mathsf{nextConf}_\delta(x, y) \wedge \mathtt{U}_{conf}(y) \wedge \mathsf{inConf}(x, z) \rightarrow \mathtt{U}_{head}(x, z) \quad \text{for } \delta \in \Delta$$

$$\mathsf{lastConf}(x) \wedge \mathsf{inConf}(x, z) \rightarrow \mathtt{U}_{head}(x, z)$$

Structures matched by this query provide direct links from each element to their configuration element, and we can thus formulate ConfCell as a UCQ and obtain the following.

Theorem 19. *Deciding containment of* LinGDlog *queries in* LinMQ^k *queries is hard for* $(k + 1)$ExpSpace.

It is not clear if this result can be extended to containments of LinMQ in LinMQ^k; the above approach does not suggest any suitable modification. In particular, the propagation of inConf in the style of a transitive closure does not work, since elements may participate in many inConf relations. On the other hand, the special constants λ in LinMQs cannot be used to refer to the current configuration, since there can be an unbounded number of configurations but only a bounded number of special constants. It is possible, however, to formulate a LinMQ Config$[x]$ that generates the required structure for a single configuration, since one can then represents the configuration by λ. We can generate arbitrary sequences of such structures by using Config$[x]$ as a nested query to that matches a regular expression firstConf (Config NextConf)* Config lastConf, where we use NextConf to express the disjunction of all $\mathsf{nextConf}_\delta$ relations. This proves the following statement.

Theorem 20. *Deciding containment of* LinMQ^2 *queries in* LinMQ^k *queries is hard for* $(k + 1)$ExpSpace.

Finally, we can also continue to use the same approach for encoding SameCell as in Section 8, without using ConfCell, while still restricting to linear Datalog (and thus to non-alternating TMs) on the left-hand side. This leads us to the following result.

Theorem 21. *Deciding containment of* LinMDlog *queries in* MQ^k *queries is hard for* $(k + 1)$ExpSpace.

We have thus established tight complexity bounds for the containment of nested GQs, while there remains a gap (of one exponential or one nesting level) for MQs.

10. CONCLUSIONS

We have studied the most expressive fragments of Datalog for which query containment is still known to be decidable today, and we have provided exact complexities for most of their query answering and query containment problems. While containment for nested queries tends to be non-elementary for unbounded nesting depth, we have shown tight exponential complexity hierarchies for the main cases that we studied. As part of our results, we have also settled a number of open problems for known query languages: the complexity of query containment for MQ and MQ^+, the complexity of query containment of Dlog in GDlog, and the expressivity of nested LinDlog.

Moreover, we have built on the recent "flag & check" approach of monadically defined queries to derive various natural extensions, which lead to new query languages with interesting complexity results. In most cases, we observed that the extension from monadic to frontier-guarded Datalog does not affect any of the complexities, whereas it might have an impact on expressivity. In contrast, the restriction to linear Datalog has the expected effects, both for query answering and for query containment.

The only case for which our results for containment complexity are not tight is when we restrict rules to be both linear and monadic: while small variations in the involved query languages lead to the expected tight bounds, this particular combination eludes our analysis. This case could be studied as part of a future program for analyzing the behavior of (nested) conjunctive regular path queries, which are also a special form of monadic, linear Datalog.

Another interesting open question is the role of constants. Our hardness proofs, especially in the nested case, rely on

	UCQ, LinMDlog, MDlog, LinGDlog, GDlog	LinMQ^k, LinGQ^k	MQ^k, GQ^k	LinMQ$^+$,MQ$^+$, LinGQ$^+$,GQ$^+$	Dlog
LinMQ	PSpace-h [13] ExpSpace [Th.16]	kExpSpace-h [Th.18] $(k+1)$ExpSpace [Th.17]	$(k+1)$ExpSpace-c [Th.21]\[Th.17]	Nonelementary [Th.18]	Undecidable [1]
LinGDlog, LinMQ^n ($n \geq 2$), LinMQ$^+$, LinGQ$^+$, LinGQ^n, LinDlog	ExpSpace-c [Th.20]\[Th.16]	$(k+1)$ExpSpace-c [Th.19,20]\[Th.17]	$(k+1)$ExpSpace-c [Th.19,20]\[Th.17]	Nonelementary [Th.19,20]	Undecidable [1]
MDlog, GDlog, MQ^n, GQ^n, MQ$^+$, GQ$^+$, Dlog	2ExpTime-c [6, 14]\ [15], [Th.10]	$(k+2)$ExpTime-c [Th.13]\[Th.11]	$(k+2)$ExpTime-c [Th.13]\[Th.11]	Nonelementary [Th.13]	Undecidable [21]

Table 1: Summary of the known complexities of query containment for several Datalog fragments; sources for each claim are shown in square brackets, using \ to separate sources for lower and upper complexity bounds, respectively

the use of constants to perform certain checks more efficiently. Without this, it is not clear how an exponential blow-up of our encoding (or the use of additional nesting levels) could be avoided. Of course, constants can be simulated if we have either predicates of higher arity or special constants as in "flag & check" queries. However, for the case of (linear) monadic Datalog without constants, we conjecture that containment complexities are reduced by one exponential each when omitting constants.

An additional direction of future research is to study problems where we ask for the *existence* of a containing query of a certain type rather than merely check containment of two given queries. The most prominent instance of this scenario is the *boundedness* problem, which asks whether a given Datalog program can be expressed by some (yet unknown) UCQ. It has been shown that this problem can be studied using tree-automata-based techniques as for query containment [15], though other approaches have been applied as well [4]. Besides boundedness, one can also ask more general questions of *rewritability*, e.g., whether some Datalog program can be expressed in monadic Datalog or in a regular path query.

11. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1994.

[2] S. Abiteboul and V. Vianu. Regular path queries with constraints. *J. Comput. Syst. Sci.*, 58(3):428–452, 1999.

[3] J.-F. Baget, M. Leclère, M.-L. Mugnier, and E. Salvat. On rules with existential variables: Walking the decidability line. *Artificial Intelligence*, 175(9–10):1620–1654, 2011.

[4] V. Bárány, B. ten Cate, and M. Otto. Queries with guarded negation. *PVLDB*, 5(11):1328–1339, 2012.

[5] V. Bárány, B. ten Cate, and L. Segoufin. Guarded negation. In L. Aceto, M. Henzinger, and J. Sgall, editors, *ICALP (2)*, volume 6756 of *Lecture Notes in Computer Science*, pages 356–367. Springer, 2011.

[6] M. Benedikt, P. Bourhis, and P. Senellart. Monadic datalog containment. In *Automata, Languages, and Programming - 39th International Colloquium, ICALP 2012*, pages 79–91, 2012.

[7] H. Björklund, W. Martens, and T. Schwentick. Optimizing conjunctive queries over trees using schema information. In E. Ochmanski and J. Tyszkiewicz, editors, *Proc. 3rdInt. Symposium on Mathematical Foundations of Computer Science*, volume 5162 of *LNCS*, pages 132–143. Springer, 2008.

[8] A. Calì, G. Gottlob, and M. Kifer. Taming the infinite chase: Query answering under expressive relational constraints. In G. Brewka and J. Lang, editors, *Proc. 11th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'08)*, pages 70–80. AAAI Press, 2008.

[9] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. Reasoning on regular path queries. *SIGMOD Record*, 32(4):83–92, 2003.

[10] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. Reasoning on regular path queries. *SIGMOD Record*, 32(4):83–92, 2003.

[11] D. Calvanese, G. D. Giacomo, and M. Y. Vardi. Decidable containment of recursive queries. *Theor. Comput. Sci.*, 336(1):33–56, 2005.

[12] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *J. of the ACM*, 28(1):114–133, 1981.

[13] S. Chaudhuri and M. Y. Vardi. On the complexity of equivalence between recursive and nonrecursive datalog programs. In *Proc. 13th Symposium on Principles of Database Systems (PODS'93)*, pages 107–116, 1994.

[14] S. Chaudhuri and M. Y. Vardi. On the equivalence of recursive and nonrecursive datalog programs. *J. Comput. Syst. Sci.*, 54(1):61–78, 1997.

[15] S. Cosmadakis, H. Gaifman, P. Kanellakis, and M. Vardi. Decidable optimization problems for database logic programs. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, STOC '88, pages 477–490, New York, NY, USA, 1988. ACM.

[16] B. Courcelle. Recursive queries and context-free graph grammars. *Theoretical Computer Science*, 78(1):217 – 244, 1991.

[17] A. Deutsch and V. Tannen. Optimization properties for classes of conjunctive regular path queries. In *Revised Papers from the 8th International Workshop on Database Programming Languages*, DBPL '01, pages 21–39, London, UK, UK, 2002. Springer-Verlag.

[18] D. Florescu, A. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. In A. O. Mendelzon and J. Paredaens, editors, *Proc. 17th Symposium on Principles of Database Systems (PODS'98)*, pages 139–148. ACM, 1998.

[19] S. Rudolph and M. Krötzsch. Flag & check: Data access with monadically defined queries. In R. Hull and W. Fan, editors, *Proc. 32nd Symposium on Principles of Database Systems (PODS'13)*, pages 151–162. ACM, 2013.

[20] W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177 – 192, 1970.

[21] O. Shmueli. Decidability and expressiveness aspects of logic queries. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '87, pages 237–249, New York, NY, USA, 1987. ACM.

APPENDIX

A. TREE AUTOMATA

We use standard definitions for two-way alternating tree automata as introduced in [15]. A regular (one-way, non-alternating) tree automaton is obtained by restricting this definition.

Tree automata run over ranked, labelled trees of some maximal arity (out-degree) f. A ranked tree can be seen a function t mapping sequences of positive natural numbers (encoding nodes in the tree) to symbols from a fixed finite alphabet (the labels of each node). Each letter of the alphabet is ranked, i.e., associated with an arity that defines how many child nodes a node labeled with this symbol should have. The domain of t, denoted Nodes(t), satisfies the following closure property: if $\boldsymbol{i} \cdot j \in \text{Nodes}(t)$, then $\boldsymbol{i} \in \text{Nodes}(t)$ and $\boldsymbol{i} \cdot k \in \text{Nodes}(t)$ for all $1 \leq k \leq j$. Given a ranked tree t, we write $\boldsymbol{i} \in \text{Nodes}(t)$ to denote an arbitrary node of t and $t(\boldsymbol{i})$ to denote the label of $\boldsymbol{i}$ in t. We denote by Trees(Σ) the set of trees over the alphabet Σ.

A two-way alternating tree automaton $\mathcal{A}$ is a tuple $\langle \Sigma, Q, Q_s, \delta, Q_e \rangle$ where

- Σ is a tree alphabet;
- Q is a set of states;
- $Q_s \subseteq Q$ is the set of initial states;
- $Q_e \subseteq Q$ is the set of accepting states;
- δ is a transition function from $Q \times \Sigma$: let $q \in Q$ be a state and $\sigma \in \Sigma$ be a letter of arity ℓ; then $\delta(q, \sigma)$ is a positive boolean combination of elements in $\{-1, 0, 1, \cdots, \ell\} \times Q$.

The numbers used in transitions encode directions, where -1 is up and 0 is stay. For example $\delta(q, \sigma) = (\langle 1, s_1 \rangle \wedge \wedge 1, s_2) \vee (\langle -1, t_3 \rangle \wedge \langle 2, t_4 \rangle)$ is an example of transition for a state q and a node labeled σ: a node labeled by σ can be in the state q iff its first child can be in the states s_1 and s_2, or its parent and its second child can be in the states s_3 and s_4, respectively.

Let t be a tree over Σ. A run τ of $\mathcal{A}$ over t is a tree labeled by elements of $Q \times \{-1, 0, 1, \cdots, f\} \times \text{Nodes}(t) \cup \{-1\}$. τ satisfies the following properties:

- τ is finite.
- The root of τ is labelled by (q_0, i, n), where q_0 is in Q_s.
- If a node v is labelled by (q, i, n) and n is not a node of t, then v is a leaf of τ.
- If a node v is labelled by (q, i, n'), n is a node of τ labelled by σ of arity l and v' is labelled by (q_1, j, n') then
  - if $j = -1$, then there exists $u \leq k$ such that $n = n'.u$
  - if $j = 0$, then $n = n'$
  - if $j \leq k$, then $n' = n.j$.
- if a node v is labelled by (q, i, n), $n \in t$ labelled by σ and the children of v are labelled by $(q_1, j_1, n_1) \cdots (q_k, j_k, n_k)$ then $\delta(q, \sigma)$ is satisfied when interpreting the sybmols $\{\langle j_1, q_1 \rangle, \cdots, \langle j_k, q_k \rangle\}$ as *true* and all other symbols as *false*.

τ is *valid* iff, for each leaf of τ labelled by (q, i, n), q is in Q_e. $\mathcal{A}$ *accepts* a tree t if there exists a valid run of t over $\mathcal{A}$. We denote by Trees($\mathcal{A}$). The set of trees accepted by $\mathcal{A}$.

A regular (one-way, non-alternating) tree automaton is a 2-way alternating tree automaton where all transitions for a symbol σ of rank ℓ are boolean formulae of the form $(\langle 1, q_{11} \rangle \wedge \ldots \wedge \langle \ell, q_{\ell 1} \rangle) \vee \ldots \vee (\langle 1, q_{1n} \rangle \wedge \ldots \wedge \langle \ell, q_{\ell n} \rangle)$ for some $n \geq 0$. In particular, directions 0 and -1 do not occur. In this case, we can represent transitions as sets of lists of states $\{\langle q_{11}, \ldots, q_{\ell 1} \rangle, \ldots, \langle q_{1n}, \ldots, q_{\ell n} \rangle\}$.

Finally, we recall two useful theorems from [15].

THEOREM 22 (THEOREM A.1 OF [15]). *Let $\mathcal{A}$ be a two-way alternating automaton. Then there exists a tree automaton $\mathcal{A}$ whose size is exponential in the size of $\mathcal{A}$ such that* Trees($\mathcal{A}$) = Trees(Σ) \ Trees($\mathcal{A}$).

THEOREM 23 (THEOREM A.2 OF [15]). *Let $\mathcal{A}$ be a two-way alternating automaton. Then there exists a tree automaton $\mathcal{A}$ whose size is exponential in the size of $\mathcal{A}$ such that* Trees($\mathcal{A}$) = Trees($\mathcal{A}$).

B. PROOFS

Proofs for Section 4

THEOREM 1. LinDlog = LinDlog$^+$.

PROOF. We will prove that any LinDlog$^+$ query can be rewritten into a LinDlog query of polynomial size. We make simplifying assumptions on the structure of the nested query which can be easily obtained by polynomial transformations and make the presentation easier: we assume that every rule body of any query occurring at any nesting depth contains at most one subquery atom (using, e.g., Proposition 2). Second, we assume that all variables and IDB predicates that are not in the same scope are appropriately renamed apart.

In order to proof our claim, we will first show that any LinDlog^2 can be rewritten into an equivalent LinDlog query. Applying the rewriting iteratively inside-out (and observing that even manyfold application can be done in polynomial total time) then allows to conclude that there is a polynomial rewriting of any LinDlog$^+$ query of arbitrary depth into a LinDlog query.

Consider a LinDlog^2 query $P = \langle \mathbb{P}, p \rangle$ and assume w.l.o.g. that every rule body of the rules contains at most one LinDlog^1 subquery. Now, going through all rules of $\mathbb{P}$ we produce the rules $\mathbb{P}'$ of the unnested but equivalent version.

Consider a rule $\rho \in \mathbb{P}$ having the shape

$$Q(x_1, \ldots, x_n) \wedge p(y_1, \ldots y_\ell) \wedge B_1 \wedge \ldots \wedge B_k \rightarrow H$$

where p is the body IDB predicate and where $Q = \langle \mathbb{Q}, q \rangle$ is a LinDlog^1 query. For any k-ary IDB predicate r inside $\mathbb{Q}$ we increase its arity by ℓ and let $\mathbb{P}'$ contain all rules of $\mathbb{Q}'$ which is obtained from the rules ρ' of $\mathbb{Q}$ by

- replacing any (head or body) IDB atom $r(z_1, \ldots, z_k)$ of ρ' by $r(z_1, \ldots, z_k, y_1, \ldots y_\ell)$ and
- in case ρ' does not contain any IDB body atom, add $p(y_1, \ldots y_\ell)$ to the body.

Further we let $\mathbb{P}'$ contain the rule

$$q(x_1, \ldots, x_n, y_1, \ldots y_\ell) \wedge \wedge B_1 \wedge \ldots \wedge B_k \rightarrow H.$$

In case of a rule $\rho \in \mathbb{P}$ having the shape

$$Q(x_1, \ldots, x_n) \wedge B_1 \wedge \ldots \wedge B_k \rightarrow H$$

we add $\mathbb{Q}$ to $\mathbb{P}'$ without change and let $\mathbb{P}'$ contain the rule

$$q(x_1, \ldots, x_n) \wedge B_1 \wedge \ldots \wedge B_k \rightarrow H.$$

In case a rule $\rho \in \mathbb{P}$ does not contain a subquery atom we simply add ρ to $\mathbb{P}'$.

It can now easily verified that $\langle \mathbb{P}, p \rangle$ and $\langle \mathbb{P}', p \rangle$ are equivalent: first it is straightforward, that $\langle \mathbb{P}, p \rangle$ is equivalent to $\langle \mathbb{P}^\flat, p \rangle$ where $\mathbb{P}^\flat$ is obtained from $\mathbb{P}$ by replacing every $Q(x_1, \ldots, x_n)$ by $q(x_1, \ldots, x_n)$ (that is, the according goal predicate) and then adding all rules from $\mathbb{Q}$ with no changes made to them. Second one can show that there is a direct correspondence between proof trees of $\langle \mathbb{P}^\flat, p \rangle$ and linearized proof trees of $\langle \mathbb{P}', p \rangle$ which yields the desired result. □

Proposition 2. *Let P be a positive query, i.e., a Boolean expression of disjunctions and conjunctions, of* LinMQ^k *queries with $k \geq 1$. Then there is a* LinMQ^k *query P' of size polynomial in P that is equivalent to P. Analogous results hold when replacing* LinMQ^k *by* MQ^k, GQ^k, *or* LinMQ^k *queries.*

Proof. We show the claim by induction, by expressing the innermost disjunctions and conjunctions of P with equivalent LinMQ^k queries of linear size. We consider positive queries without existential quantifiers (i.e., where all variables are answer variables), but the inner LinMQ^k may use existential quantifiers.

Let $P[\boldsymbol{x}] = P_1[\boldsymbol{x_1}] \vee \ldots \vee P_n[\boldsymbol{x_n}]$ be a disjunction of LinMQ^k queries. Each query P_i is of the form $\exists \boldsymbol{z_i}.P_i'[\boldsymbol{x_i'}]$, where $\boldsymbol{x_i'}$ is the list of free variables of P_i' (corresponding to constants λ), and $\boldsymbol{z_i}$ contains exactly those variables of $\boldsymbol{x_i'}$ that do not occur in $\boldsymbol{x_i}$. We assume without loss of generality that $\boldsymbol{z_i}$ is disjoint from $\boldsymbol{z_j}$ if $i \neq j$, and that each P_i' uses a unique set of IDBs that does not occur in other queries. We consider queries $\bar{P}_i$ obtained by replacing the special constant that represents a variable $x_j \in \boldsymbol{x}$ by the special constant λ_j (assumed to not occur in P yet). Thus, the queries $\bar{P}_i$ share special constants exactly where queries P_1 share variables. We can now define the LinMQ^k P' as $\exists \boldsymbol{z_1} \ldots \boldsymbol{z_n}.\bar{P}_1 \cup \ldots \cup \bar{P}_n$, where we assume that the correspondence of special constants to free variables is such that the existential quantifiers refer to the same variables as before.

Let $P[\boldsymbol{x}] = P_1[\boldsymbol{x_1}] \wedge \ldots \wedge P_n[\boldsymbol{x_n}]$ be a conjunction of LinMQ^k queries. Let $P_i = \exists \boldsymbol{z_i}.P_i'[\boldsymbol{x_i'}]$ as before, and let U_i for $i \in \{1, \ldots, n-1\}$ be fresh IDB predicates. The queries $\bar{P}_i$ are defined as before by renaming special constants to reflect shared variables. For each $i \in \{1, \ldots, n\}$, the set of rules $\hat{P}_i$ is obtained from $\bar{P}_i$ as follows: if $i < n$, then every rule $\varphi \rightarrow \mathsf{hit} \in \bar{P}_i$ is replaced by the rule $\varphi \rightarrow \mathsf{U}_i(\lambda_1)$, where λ_1 is a fixed special constant in the queries; if $i > 1$, then every rule $\varphi \rightarrow \psi \in \bar{P}_i$ where φ does not contain an IDB predicate is replaced by the rule $\varphi \wedge \mathsf{U}_{i-1}(\lambda_1) \rightarrow \psi$, where λ_1 is as before. The LinMQ^k P' is defined as $\exists \boldsymbol{z_1} \ldots \boldsymbol{z_n}.\hat{P}_1 \cup \ldots \cup \hat{P}_n$.

These constructions lead to equivalent LinMQ^k queries of linear size, so the claim follows by inductions. The cases for MQ^k, GQ^k, and LinMQ^k follow from the same constructions (note that, without the requirement of linearity, a simpler construction is possible in the case of conjunctions). □

Theorem 3. *The combined complexity of evaluating* GQ *queries over a database instance is* NP*-complete. The same holds for* GDlog *queries. The combined complexity of evaluating* GQ^+ *queries is* PSpace*-complete. The data complexity is* P*-complete for* GDlog, GQ, *and* GQ^+.

Proof. The lower bounds are immediate from the matching complexities for MQ and MQ^+ queries, respectively [19].

First, we prove that checking if a tuple is an answer of a GQ over a database instance $\mathcal{I}$ is in NP for combined complexity. Let $\mathcal{I}$ be an instance, let P be a GQ with frontier guarded rules $\mathbb{P}$, and let $\boldsymbol{\delta}$ be be a candidate answer for P as in Definition 1.

Since each rule in $\mathbb{P}$ is frontier-guarded, each intentional fact that is derived when checking the answer follows from the application of one particular rule, instantiated to match one particular (guard) EDB fact in the body. Therefore, the number of IDB facts that can be derived is polynomially bounded in the size of $\mathcal{I}$ and $\mathbb{P}$.

Thus, for every derivation of $\mathbb{P}$, only a polynomial number of rule applications are necessary, since it is enough to derive each IDB fact once. It is clear that one can guess such a derivation, where we guess, for each derivable IDB fact, one specific rule instance by which it is derived. The correctness of this guess can be checked in polynomial time, showing that the problem can be solved in NP.

We now show that checking an answer of a GQ^+ over an instance $\mathcal{I}$ is in PSpace. Let $\mathcal{I}$ be an instance, let P be a GQ^k with frontier guarded rules $\mathbb{P}$ (that may contain subqueries), and let $\boldsymbol{\delta}$ be be a candidate answer for P as in Definition 1. We demonstrate by induction on k that checking if $\boldsymbol{\delta}$ is a solution for P w.r.t. $\mathcal{I}$ is in NPSpace. For the induction base, the claim follows from the above result for GQs.

For the induction step, using the same argument as before, we can see that the number of IDB facts that can be derived by $\mathbb{P}$ is still polynomial. Therefore, we can again guess a polynomial derivation as before, though the rule instances now may refer to subqueries of smaller nesting depth. By the induction hypothesis, whenever we need to verify the applicability of such a rule, we can use an NPSpace algorithm for the nested query. The overall number of such checks is polynomial, yielding the overall NPSpace algorithm. The result follows since NPSpace =PSpace [20].

The fact that query evaluation is in P for data complexity is immediate from the fact our queries can be expressed in Datalog, which is known to have this data complexity. A direct proof is also obtained by observing that the number of possible derivation sequences that the above algorithms need to consider is in itself polynomial in $\mathcal{I}$ if P is fixed, so that the algorithms themselves are already in P for data complexity. □

Theorem 4. *The combined complexity of evaluating* LinMQ *queries over a database instance is* NP*-complete. The same holds for* LinGDlog *and* LinGQ*. The combined complexity of evaluating* LinMQ$^+$ *queries is* PSpace*-complete. The same holds for* LinGQ$^+$.

The data complexity is NLogSpace*-complete for all of these query languages.*

Proof. The claimed NP-completeness is immediate. Hardness follows from the hardness of CQ query answering. Membership follows from the membership of GQ.

The claimed membership in PSpace follows from the PSpace-membership of LinDlog; note that this uses Theorem 1. Hardness for LinGQ$^+$ follows from the hardness for LinMQ$^+$, which we show by modifying the PSpace-hardness proof for monadically defined queries from [19].

We show the result by providing a reduction from the validity problem of quantified Boolean formulae (QBFs). We recap that for any QBF, it is possible to construct in polynomial time an equivalent QBF that has the specific shape

$$Q_1 x_1 Q_2 x_2 \dots Q_n x_n \bigvee_{L\in\mathcal{L}} \bigwedge_{\ell\in L} \ell,$$

with $Q_1, \dots Q_n \in \{\exists, \forall\}$ and $\mathcal{L}$ being a set of sets of literals over the propositional variables $x_1, \dots, x_n$. In words, we assume our QBF to be in prenex form with the propositional part of the formula in disjunctive normal form. For every literal set $L = \{x_{k_1}, \dots, x_{k_i}, \neg x_{k_{i+1}}, \dots, \neg x_{k_j}\}$, we now define the n-ary FCP $p_L = \{t(\lambda_{k_1}) \wedge \dots \wedge t(\lambda_{k_i}) \wedge f(\lambda_{k_{i+1}}) \wedge \dots \wedge f(\lambda_{k_j}) \rightarrow \mathsf{hit}\}$. Moreover, we define the n-ary FCP $p_{\mathcal{L}} = \{p_L(\lambda_1, \dots, \lambda_n) \rightarrow \mathsf{hit} \mid L \in \mathcal{L}\}$. Letting $p_{\mathcal{L}} = p_n$ we now define FCPs $p_{n-1} \dots p_0$ in descending order. If $Q_i = \exists$, then the $i-1$-ary FCP p_{i-1} is defined as the singleton rule set $\{p_i(\lambda_1, \dots, \lambda_{i-1}, y) \rightarrow \mathsf{hit}\}$. In case $Q_i = \forall$, we let p_{i-1} contain the rules

$$\begin{aligned} f(x) &\rightarrow \mathtt{U}_?(x) \\ \mathtt{U}_!(x) \wedge f(x) \wedge t(y) &\rightarrow \mathtt{U}_?(y) \\ \mathtt{U}_!(x) \wedge t(x) &\rightarrow \mathsf{hit} \\ \mathtt{U}_?(x) \wedge p_i(\lambda_1, \dots, \lambda_{i-1}, x) &\rightarrow \mathtt{U}_!(x) \end{aligned}$$

Note that p_0 is a Boolean LinMQ$^+$ query the size of which is polynomial in the size of the input QBF.

Now, let D be the database containing the two individuals 0 and 1 as well as the facts $f(0)$ and $t(1)$. We now show that the considered QBF is true exactly if $D \models p_0()$. To this end, we first note that by construction the extension of p_L contains exactly those n-tuples $\langle\delta_1, \dots, \delta_n\rangle$ for which the corresponding truth value assignment *val*, sending x_i to **true** iff $\delta_i = 1$, makes the formula $\bigwedge_{\ell\in L} \ell$ true. In the same way, the extension of $p_{\mathcal{L}}$ represents the set of truth value assignments satisfying $\bigvee_{L\in\mathcal{L}} \bigwedge_{\ell\in L} \ell$. Then, by descending induction, we can show that the extensions of p_i encode the assignments to free propositional variables of the subformula $Q_{i+1}x_{i+1} \dots Q_n x_n \bigvee_{L\in\mathcal{L}} \bigwedge_{\ell\in L} \ell$ that make this formula true. Consequently, p_0 has a nonempty extension if the entire considered QBF is true.

Finally, the NLogSpace-completeness for data complexity is again immediate, where the upper bound is obtained from LinDlog, and the lower bound follows from the well-known hardness of reachability queries, which can be expressed in LinMDlog. □

Proofs for Section 6

Proposition 7. *There is an automaton $\mathcal{A}_{P,\rho}$ that accepts exactly the annotated matching trees for ρ and $\mathbb{P}$, and which is exponential in the size of ρ and $\mathbb{P}$.*

Proof. We first construct an automaton $\mathcal{A}'_{P,\rho}$ that accepts matching trees where each node is additionally annotated by a partial mapping of the form $\mathsf{Var}(\rho) \rightarrow \mathcal{V}_{\mathbb{P}}$ (called $\mathsf{Var}(\rho)$-*label*), such that: every special variable $x \in \mathsf{Var}(\rho)$ occurs in at least one $\mathsf{Var}(\rho)$-label, and whenever a variable $x \in \mathsf{Var}(\rho)$ occurs in two, it is mapped to the same variable and both variable occurrences are connected. Note that this is essentially the same condition that we imposed for $\boldsymbol{\lambda}$-annotations.

The intersection of tree automata can be computed in polynomial time. We can therefore construct automata to check part of the conditions for (annotated) matching trees to simplify the definitions. We first construct an automaton $\mathcal{A}_x$ for checking the condition on $\mathsf{Var}(\rho)$-labels for one variable $x \in \mathsf{Var}(\rho)$. We define $\mathcal{A}_x = \langle\Sigma, Q_x, Q_x^s, \delta_x, Q_x^e\rangle$, where the alphabet Σ consists of quadruples of proof-tree labels (from $\mathcal{R}_{\mathbb{P}}$), $\boldsymbol{\lambda}$-labels, p-labels, and $\mathsf{Var}(\rho)$-labels. The state set Q_x is $\{a, b, \mathsf{accept}\} \cup \{q_v \mid v \in \mathcal{V}_{\mathbb{P}}\}$, signifying that the current node is *a*bove the first node annotated with a mapping for x, *b*elow or *b*esides any nodes that were annotated with a mapping for x, or at a node where x is mapped to a variable v. That start-state set is $Q_x^s = \{a\} \cup \{q_v \mid v \in \mathcal{V}_{\mathbb{P}}\}$; the end-state set if $Q_x^e = \{\mathsf{accept}\}$.

Consider a rule $\rho' \in \mathcal{R}_{\mathbb{P}}$ of the form $r_1(\boldsymbol{v_1}) \wedge \dots \wedge r_n(\boldsymbol{v_n}) \wedge h_1(\boldsymbol{w_1}) \wedge \dots \wedge h_m(\boldsymbol{w_m}) \rightarrow h(\boldsymbol{v})$, where r_i are EDB predicates and $h_{(i)}$ are IDB predicates. For the case that $m > 0$, there is a transition $\langle q_1, \dots, q_m\rangle \in \delta(q, \langle\rho', _, _, \nu\rangle)$ exactly if the following conditions are satisfied:

- if $q = a$ and $\nu(x)$ is undefined, then $q_i = a$ for one $1 \le i \le m$ and $q_j = b$ for all $1 \le j \le m$ with $i \ne j$;
- if $q = q_v$ and $\nu(x) = v$, then $q_i = q_v$ for all $1 \le i \le m$ such that v occurs in $\boldsymbol{w_i}$ and $q_i = b$ for all other i;

- if $q = b$ and $\nu(x)$ is undefined, then $q_i = b$ for all $1 \leq i \leq m$.

For the case $m = 0$, there is a transition $\langle\mathsf{accept}\rangle \in \delta(q, \langle\rho', _, _, \nu\rangle)$ exactly if:

- if $q = q_v$ and $\nu(x) = v$;
- if $q = b$ and $\nu(x)$ is undefined.

It is easy to check that the automaton $\mathcal{A}_x$ satisfies the required condition. Now an automaton for checking the condition on $\mathsf{Var}(\rho)$-labels can be constructed as the intersection $\mathcal{A}'_{\mathsf{Var}(\rho)} = \bigcap_{x\in\mathsf{Var}(\rho)} \mathcal{A}_x$. The automaton $\mathcal{A}'_{\boldsymbol{\lambda}}$ for checking the condition on $\boldsymbol{\lambda}$-labels is constructed in a similar fashion. Likewise, an automaton $\mathcal{A}'_p$ for checking the condition on p-labels is easy to define.

It remains to construct an automaton for checking the conditions (a)–(d) of Definition 4. To do this, we interpret the $\mathsf{Var}(\rho)$-labels and $\boldsymbol{\lambda}$-labels as partial specifications of the required mapping ν. Condition (a) further requires that $\nu(\boldsymbol{x}) = \boldsymbol{v}$, i.e., that the $\mathsf{Var}(\rho)$-label at the unique node annotated with $p(\boldsymbol{v})$ contains this mapping. It is easy to verify this with an automaton $\mathcal{A}'_{(a)}$. Together, $\mathcal{A}'_{(a)}$, $\mathcal{A}'_{\boldsymbol{\lambda}}$, and $\mathcal{A}'_{\mathsf{Var}(\rho)}$ provide a consistent variable mapping that respects the p-label (a) and the connectedness of variable occurrences, i.e., (c) and (d). To check the remaining condition (b), we use an automaton $\mathcal{A}'_{(b)}$.

The automaton for (b) will use auxiliary markers to record which atoms have been matched in the current node and how exactly this was done. We record such a match as a partial function from atoms $q(\boldsymbol{z}) \in \varphi$ to instances $q(\boldsymbol{w})$ of such atoms using variables $\boldsymbol{w} \subseteq \mathcal{V}_{\mathbb{P}}$. The set of all such partial functions is denoted $\mathsf{Match}_{\varphi,\mathbb{P}}$. Note that this set is exponential (not double exponential).

We now define $\mathcal{A}'_{(b)} = \langle\Sigma, Q, Q_s, \delta, Q_e\rangle$ where Σ is as for $\mathcal{A}_x$ above. The set of states Q is $\{\mathsf{accept}\} \cup (2^{\varphi} \times \mathsf{Match}_{\varphi,\mathbb{P}})$, where elements from 2^{φ} encode the subset of φ that should be witnessed at or below the current node, and the elements from $\mathsf{Match}_{\varphi,\mathbb{P}}$ encode atoms that must be matched at the current node with their respective instantiations. The start-state set Q_s is $\{\langle\varphi, \mu\rangle \mid \mu \in \mathsf{Match}_{\varphi,\mathbb{P}}\}$; the end-state set Q_e is $\{\mathsf{accept}\}$. The transition function δ is defined as follows. Consider a rule $\rho' \in \mathcal{R}_{\mathbb{P}}$ of the form $r_1(\boldsymbol{v_1}) \wedge \ldots \wedge r_n(\boldsymbol{v_n}) \wedge h_1(\boldsymbol{w_1}) \wedge \ldots \wedge h_m(\boldsymbol{w_m}) \rightarrow h(\boldsymbol{v})$, where r_i are EDB predicates and $h_{(i)}$ are IDB predicates. For the case $m > 0$, there is a transition $\langle\langle\beta_1, \mu_1\rangle, \ldots, \langle\beta_m, \mu_n\rangle\rangle \in \delta(\langle\beta, \mu\rangle, \langle\rho', \nu_{\boldsymbol{\lambda}}, _, \nu_{\mathsf{Var}(\rho)}\rangle)$ exactly if the set $\beta \subseteq \varphi$ can be partitioned into sets $\beta', \beta_1, \ldots, \beta_m$ such that $(\nu_{\boldsymbol{\lambda}} \cup \nu_{\mathsf{Var}(\rho)})(\beta') = \mu(\beta')$ and $\mu(\beta') \subseteq \{r_1(\boldsymbol{v_1}), \ldots, r_n(\boldsymbol{v_n})\}$. The element μ_i of successor states can be chosen freely; the validity of the choice will be checked later. For the case $m = 0$, there is a transition $\langle\mathsf{accept}\rangle \in \delta(\langle\beta, \mu\rangle, \langle\rho', \nu_{\boldsymbol{\lambda}}, _, \nu_{\mathsf{Var}(\rho)}\rangle)$ exactly if $(\nu_{\boldsymbol{\lambda}} \cup \nu_{\mathsf{Var}(\rho)})(\beta) = \mu(\beta)$ and $\mu(\beta) \subseteq \{r_1(\boldsymbol{v_1}), \ldots, r_n(\boldsymbol{v_n})\}$. In fact, the information from $\mathsf{Match}_{\varphi,\mathbb{P}}$ is not strictly necessary to define the transition, since the relevant elements μ are always determined by other choices in the transition. However, having this information explicit will be important in later proofs.

The automaton $\mathcal{A}'_{P,\rho}$ is obtained as the intersection $\mathcal{A}'_{\mathsf{Var}(\rho)} \cap \mathcal{A}'_{\boldsymbol{\lambda}} \cap \mathcal{A}'_p \cap \mathcal{A}'_{(a)} \cap \mathcal{A}'_{(b)}$. It is easy to verify that it accepts exactly the $\mathsf{Var}(\rho)$-annotated matching trees. Note that $\mathcal{A}'_{P,\rho}$ is exponential in size, already due to the exponentially large alphabet Σ. Now the required automaton $\mathcal{A}_{P,\rho}$ is obtained by "forgetting" the $\mathsf{Var}(\rho)$-label in transitions of $\mathcal{A}'_{P,\rho}$. This projection operation for tree automata is possible with a polynomial increase in size: every state of $\mathcal{A}_{P,\rho}$ is a pair of a state of $\mathcal{A}'_{P,\rho}$ and a $\mathsf{Var}(\rho)$-label; transitions of $\mathcal{A}_{P,\rho}$ are defined as for $\mathcal{A}'_{P,\rho}$, but keeping $\mathsf{Var}(\rho)$-label information in states and introducing transitions for all possible $\mathsf{Var}(\rho)$-labels in child nodes. □

Proposition 8. *There is an alternating 2-way tree automaton $\mathcal{A}^+_{P,\rho,\boldsymbol{v}}$ that is polynomial in the size of $\mathcal{A}_{P,\rho}$ such that, whenever $\mathcal{A}_{P,\rho}$ accepts a matching tree T that has the p-annotation $p(\boldsymbol{v})$ on node e, then $\mathcal{A}^+_{P,\rho,\boldsymbol{v}}$ has an accepting run that starts from the corresponding node e' on the tree T' that is obtained by removing the p-annotation from T.*

Proof. Using alternating 2-way automata, we can traverse a tree starting from any node, visiting each node once. To control the direction of the traversal, we create multiple copies of each state q: states q_{down} are processed like normal states in $\mathcal{A}_{P,\rho}$, states q_{up} use an inverted transition of $\mathcal{A}_{P,\rho}$ to move up the tree into a state $q_{\sigma,i}$; these auxiliary states are used to check that the label of the upper node is actually σ and to start new downwards processes for all child nodes other than the one (i) that we came from.

To ensure that the constructed automaton $\mathcal{A}^+_{P,\rho,\boldsymbol{v}}$ simulates the behavior of $\mathcal{A}_{P,\rho}$ in case the annotation $p(\boldsymbol{v})$ is found, we eliminate all transitions that mention other p-annotations. Moreover, we assume without loss of generality that the states of $\mathcal{A}_{P,\rho}$ that allow a transition mentioning $p(\boldsymbol{v})$ cannot be left through any other transition; this can always be ensured by duplicating states and using them exclusively for one kind of transition. Let Q_p be the set of states of $\mathcal{A}_{P,\rho}$ that admit (only) transitions mentioning $p(\boldsymbol{v})$. Let $\mathcal{A}'_{P,\rho} = \langle\Sigma', Q, Q_s, \delta', Q_e\rangle$ denote the automaton over the alphabet Σ' of $\boldsymbol{\lambda}$-annotated proof trees (without p-annotations), with the same (start/end) states as $\mathcal{A}_{P,\rho}$, and where δ' is defined based on the transition function δ of $\mathcal{A}_{P,\rho}$ as follows: $\delta'(\langle\rho', M\rangle)$ is the union of all sets of the form $\delta(\langle\rho', \boldsymbol{\lambda}\text{-label}, p\text{-label}\rangle)$ where p-label is either $p(\boldsymbol{v})$ or empty. By this construction, there is a correspondence between the accepting runs of $\mathcal{A}_{P,\rho}$ over trees where one node e is annotated with $p(\boldsymbol{v})$ and accepting runs of $\mathcal{A}'_{P,\rho}$ (on trees without p-annotations) for which the node e is visited in some state of Q_p.

Let s be the maximal out-degree of proof trees for P, i.e., the maximal number of IDB atoms in bodies of P. The state set Q^+ of $\mathcal{A}^+_{P,\rho,\boldsymbol{v}}$ is given by the disjoint union $\{q_{\mathsf{up}} \mid q \in Q\} \cup \{q_{\sigma,i} \mid q \in Q, \sigma \in \Sigma, 1 \leq i \leq s\} \cup \{q_{\mathsf{down}} \mid q \in Q\} \cup \{\mathsf{start}, \mathsf{accept}\}$. The start-state set is $Q^+_s = \{\mathsf{start}\}$ and the end-state set is $Q^+_e = \{\mathsf{accept}\} \cup \{q_{\mathsf{down}} \mid q \in Q_e\}$.

Transitions of $\mathcal{A}^+_{P,\rho,\boldsymbol{v}}$ are defined as follows:

- For all $\sigma \in \Sigma$, let $\delta^+(\mathsf{start}, \sigma)$ be the disjunction of all formulae $\langle 0, q_{\mathsf{up}}\rangle \wedge \langle 0, q_{\mathsf{down}}\rangle$ where $q \in Q_p$.
- For states q_{down} and $\sigma \in \Sigma$, let $\delta^+(q_{\mathsf{down}}, \sigma)$ be the disjunction of all formulae $\langle 1, q^1_{\mathsf{down}}\rangle \wedge \ldots \wedge \langle m, q^m_{\mathsf{down}}\rangle$ for which $\mathcal{A}'_{P,\rho}$ has a transition $\langle q^1, \ldots, q^m\rangle \in \delta'(q, \sigma)$.
- For states q_{up} and $\sigma \in \Sigma$, let $\delta^+(q_{\mathsf{up}}, \sigma)$ be the disjunction of all formulae $\langle -1, q'_{\sigma',i}\rangle$ for which $\mathcal{A}'_{P,\rho}$ has a transition $\langle q^1, \ldots, q^{i-1}, q, q^{i+1}, \ldots, q^m\rangle \in \delta'(q', \sigma')$ and the current node is the ith child of its parent (we can assume that this information is encoded in the labels σ, even for basic proof trees, which increases the alphabet only linearly; we omit this in our definitions since it would clutter all other parts of our proof without need).
- For states $q_{\sigma,i,q'}$, let $\delta^+(q_{\sigma,i,q'}, \sigma)$ be the disjunction of all formulae $\langle 0, q_{\mathsf{up}}\rangle \wedge \langle 1, q^1_{\mathsf{down}}\rangle \wedge \ldots \wedge \langle i-1, q^{i-1}_{\mathsf{down}}\rangle \wedge \langle i+1, q^{i+1}_{\mathsf{down}}\rangle \wedge \ldots \wedge \langle m, q^m_{\mathsf{down}}\rangle$ for which $\mathcal{A}'_{P,\rho}$ has a transition $\langle q^1, \ldots, q^{i-1}, q', q^{i+1}, q^m\rangle \in \delta'(q, \sigma)$.
- For all starting states $q \in Q_s$ of $\mathcal{A}'_{P,\rho}$ and $\sigma \in \Sigma$, let $\delta(q_{\mathsf{up}}, \sigma) = \langle 0, \mathsf{accept}\rangle$.

It is not hard to verify that $\mathcal{A}^+_{P,\rho,\boldsymbol{v}}$ has the required properties. □

Proposition 9. *For a* Dlog *query P and a* GQ *query P' with special constants $\boldsymbol{\lambda}$, there is an alternating 2-way automaton $\mathcal{A}^+_{P\sqsubseteq P'}$ of exponential size that accepts the $\boldsymbol{\lambda}$-annotated proof trees of P that encode expansion trees with $\boldsymbol{\lambda}$ assignments for which P' has a match.*

Proof. Let P' be the set $\{\rho_1, \ldots, \rho_\ell\}$. For every IDB predicate p, let P'_p denote the set of rules in P' with head predicate p (possibly hit). Without loss of generality, we assume that distinct rules use distinct sets of variables. For every frontier-guarded rule ρ', let $\mathsf{guard}(\rho')$ be a fixed EDB atom that acts as a guard in this rule, i.e., an atom that refers to all variables in the head of ρ'.

Consider a rule $\rho' \in P'$ with IDB atoms $q_1(\boldsymbol{t_1}), \ldots, q_m(\boldsymbol{t_m})$ in its body. We construct new rules from ρ' by replacing each atom $q_i(\boldsymbol{t_i})$ with a guard atom $\mathsf{guard}(\rho'_i)$, suitably unified. Formally, assume that there are rules $\rho'_i \in P'_{q_i}$ with head $q_i(\boldsymbol{s_i})$ and a substitution θ that is a most general unifier for the problems $\boldsymbol{t_i}\theta = \boldsymbol{s_i}\theta$, for all $i \in \{1, \ldots, m\}$, and that maps every variable in ρ'_i that does not occur in the head to a globally fresh variable. Then the *guard expansion* of ρ' for $(\rho'_i)_{i=1}^m$ and θ is the rule that is obtained from $\rho'\theta$ by replacing each body atom $q_i(\boldsymbol{t_i})\theta$ by $\mathsf{guard}(\rho'_i)\theta$. By construction, two distinct atoms $\mathsf{guard}(\rho'_i)\theta$ and $\mathsf{guard}(\rho'_j)\theta$ do not share variables, unless at positions that correspond to head variables in rules ρ'_i and ρ'_j. The atoms $\mathsf{guard}(\rho'_i)\theta$ in a guard expansion are called *replacement guards*. We consider two guard expansions to be equivalent if they only differ in the choice of the most general unifier. Let $\mathsf{Guard}(\rho')$ be the set of all guard expansions of $\rho' \in P'$, i.e., a set containing one representative of each class of equivalent guard expansions. $\mathsf{Guard}(\rho')$ is exponential since there are up to $|P'|^m$ non-equivalent guard expansions for a rule with m IDB atoms.

The automaton $\mathcal{A}^+_{P\sqsubseteq P'}$ is constructed as follows. For every guard expansion $\rho_g \in \bigcup_{\rho' \in P'} \mathsf{Guard}(\rho')$ and every list $\boldsymbol{v}$ of proof-tree variables of the arity of the head of ρ_g, consider the alternating 2-way tree automaton $\mathcal{A}^+_{P,\rho_g,\boldsymbol{v}}$ of Proposition 8. We assume w.l.o.g. that the state sets of these automata are mutually disjoint. Let $\mathcal{A}^+_{P\sqsubseteq P'} = \langle \Sigma, Q, Q_s, \delta, Q_e\rangle$. As before, Σ consists of pairs of a rule instance from $\mathcal{R}_\mathbb{P}$ and a partial mapping of $\boldsymbol{\lambda}$ to $\mathcal{V}_\mathbb{P}$. The state set Q is the disjoint union of all state sets of the automata of form $\mathcal{A}^+_{P,\rho_g,\boldsymbol{v}}$. The start-state set Q_s is the disjoint union of all start-state sets of automata $\mathcal{A}^+_{P,\rho_g,\boldsymbol{v}}$ for which ρ_g is a guard expansion of a rule with head hit (and $\boldsymbol{v}$ is the empty list). The end-state set Q_e is the disjoint union of all end-state sets of automata $\mathcal{A}^+_{P,\rho_g,\boldsymbol{v}}$.

The transition function δ is defined as follows. By the construction in Proposition 7, each state q in the automaton $\mathcal{A}_{P,\rho}$ encodes a partial mapping $\mathsf{match}(q)$ from body atoms of ρ to instantiated atoms that use variables from $\mathcal{V}_\mathbb{P}$, which are matched at the current tree node. This information is preserved through alphabet projections, intersections, and even through the construction in Proposition 8. We can therefore assume that each state q of $\mathcal{A}^+_{P\sqsubseteq P'}$ is associated with a partial mapping $\mathsf{match}(q)$.

For every state $q \in Q_{P,\rho_g,\boldsymbol{v}}$ and every $\sigma \in \Sigma$, we define $\delta(q, \sigma) = \delta_{P,\rho_g,\boldsymbol{v}}(q, \sigma) \wedge \psi$, where ψ defined as follows. For every replacement guard atom α of ρ_g for which $\mathsf{match}(q)(\alpha)$ is defined, we consider the formula $\psi_\alpha = \langle 0, q_1\rangle \vee \ldots \vee \langle 0, q_\ell\rangle$, where

- $\alpha = \mathsf{guard}(\rho')\theta$ for some rule ρ' and substitution θ;
- $\mathsf{match}(q)(\alpha) = \alpha\theta'$ for some substitution θ';
- $q_1, \ldots q_\ell$ are the start states of the automaton $\mathcal{A}_{P,\rho',z\theta\theta'}$ where $p(\boldsymbol{z})$ is the head of ρ'.

Now ψ is the conjunction of all formulae ψ_α thus defined. □

Proofs for Section 7

Lemma 12. *Consider an ATM $\mathcal{M}$, and queries as in Definition 5, including $\mathsf{SameCell}[x, y]$, that are MQ^k queries for some $k \geq 0$. There is a MQ^k query $P[x]$, polynomial in the size of $\mathcal{M}$ and the given queries, such that the following hold.*

- *For every accepting run of $\mathcal{M}$ in space s, there is some database instance $\mathcal{I}$ with some element c that encodes the run, such that $c \notin P^\mathcal{I}$.*
- *If an element c of $\mathcal{I}$ encodes a tree of quasi-configurations of $\mathcal{M}$ in space s, and if $c \notin P^\mathcal{I}$, then c encodes an accepting run of $\mathcal{M}$ in space s.*

Moreover, if all input queries are in LinMQ^k, then so is P.

Proof. We construct P from all (polynomially many) positive queries obtained by instantiating the query patterns in Figure 4. Since P needs to be a unary query with variable x, we extend every positive query that does not contain x

(1) Unique head marker and correct left/right head markers:

$$\mathsf{Head}(y, p_1) \wedge \mathsf{NextCell}(y, z) \wedge \mathsf{Head}(z, p_2) \quad \text{where } \langle p_1, p_2\rangle \in \{\langle h,h\rangle, \langle h,l\rangle, \langle r,h\rangle, \langle r,l\rangle\}$$
$$\mathsf{Head}(y, h) \wedge \mathsf{Head}(y, p) \quad \text{where } p \in \{r, l\}$$

(2) Unique start configuration:

$$\mathsf{FirstConf}(x, y) \wedge \mathsf{State}_q(y) \quad \text{where } q \neq q_s$$
$$\mathsf{FirstConf}(x, y) \wedge \mathsf{FirstCell}(y, z) \wedge \mathsf{Head}(z, p) \quad \text{where } p \in \{l, r\}$$
$$\mathsf{FirstConf}(x, y) \wedge \mathsf{ConfCell}(y, z) \wedge \mathsf{Symbol}(z, c_\sigma) \quad \text{where } \sigma \neq \square$$

(3) Valid, uniquely defined transitions:

$$\mathsf{State}_q(y) \wedge \mathsf{Head}(z, h) \wedge \mathsf{ConfCell}(y, z) \wedge \mathsf{Symbol}(z, c_\sigma) \wedge \mathsf{NextConf}_\delta(y, y') \wedge \mathsf{State}_{q'}(y') \wedge \mathsf{ConfCell}(y', z') \wedge \mathsf{SameCell}(z', z) \wedge \mathsf{Symbol}(z', c_{\sigma'})$$
$$\text{where } \delta = \langle q_1, \sigma_1, q_2, \sigma_2, d\rangle \text{ with } q_1 \neq q \text{ or } \sigma_1 \neq \sigma \text{ or } q_2 \neq q' \text{ or } \sigma_2 \neq \sigma'$$

(4) Unique end state:

$$\mathsf{LastConf}(y) \wedge \mathsf{State}_q(y) \quad \text{where } q \neq q_e$$

(5) Memory:

$$\mathsf{ConfCell}(y_1, x_1) \wedge \mathsf{Head}(x_1, r) \wedge \mathsf{Symbol}(x_1, c_\sigma) \wedge \mathsf{NextConf}_\delta(y_1, y_2) \wedge \mathsf{ConfCell}(y_2, x_2) \wedge \mathsf{SameCell}(x_1, x_2) \wedge \mathsf{Symbol}(x_2, c_{\sigma'}) \quad \text{where } \sigma \neq \sigma'$$
$$\mathsf{ConfCell}(y_1, x_1) \wedge \mathsf{Head}(x_1, l) \wedge \mathsf{Symbol}(x_1, c_\sigma) \wedge \mathsf{NextConf}_\delta(y_1, y_2) \wedge \mathsf{ConfCell}(y_2, x_2) \wedge \mathsf{SameCell}(x_1, x_2) \wedge \mathsf{Symbol}(x_2, c_{\sigma'}) \quad \text{where } \sigma \neq \sigma'$$

(6) Head movement:

$$\mathsf{ConfCell}(y_1, x_1) \wedge \mathsf{Head}(x_1, h) \wedge \mathsf{NextConf}_\delta(y_1, y_2) \wedge \mathsf{ConfCell}(y_2, x_2) \wedge \mathsf{SameCell}(x_1, x_2) \wedge \mathsf{NextCell}(x_2, x_2') \wedge \mathsf{Head}(x_2', p) \quad \text{where } \delta = \langle q_1, \sigma_1, q_2, \sigma_2, \text{right}\rangle \text{ and } p \in \{r, l\}$$
$$\mathsf{ConfCell}(y_1, x_1) \wedge \mathsf{Head}(x_1, h) \wedge \mathsf{NextConf}_\delta(y_1, y_2) \wedge \mathsf{ConfCell}(y_2, x_2) \wedge \mathsf{SameCell}(x_1, x_2) \wedge \mathsf{LastCell}(x_2) \wedge \mathsf{Head}(x_2, p) \quad \text{where } \delta = \langle q_1, \sigma_1, q_2, \sigma_2, \text{right}\rangle \text{ and } p \in \{r, l\}$$
$$\mathsf{ConfCell}(y_1, x_1) \wedge \mathsf{Head}(x_1, h) \wedge \mathsf{NextConf}_\delta(y_1, y_2) \wedge \mathsf{ConfCell}(y_2, x_2) \wedge \mathsf{SameCell}(x_1, x_2) \wedge \mathsf{NextCell}(x_2', x_2) \wedge \mathsf{Head}(x_2', p) \quad \text{where } \delta = \langle q_1, \sigma_1, q_2, \sigma_2, \text{left}\rangle \text{ and } p \in \{r, l\}$$
$$\mathsf{ConfCell}(y_1, x_1) \wedge \mathsf{Head}(x_1, h) \wedge \mathsf{NextConf}_\delta(y_1, y_2) \wedge \mathsf{ConfCell}(y_2, x_2) \wedge \mathsf{SameCell}(x_1, x_2) \wedge \mathsf{FirstCell}(z, x_2) \wedge \mathsf{Head}(x_2, p) \quad \text{where } \delta = \langle q_1, \sigma_1, q_2, \sigma_2, \text{left}\rangle \text{ and } p \in \{r, l\}$$

Figure 4: Queries to construct a containment encoding as in Lemma 12

with the atom $\mathsf{FirstConf}[x, x']$ (omitted for space reasons in Figure 4). By Proposition 2 we can express the disjunctions of all the positive queries in Figure 4 as a LinMQ^k $P[x]$ of polynomial size (for $k = 0$ it is a UCQ).

If an element c in a database instance $\mathcal{I}$ encodes an accepting run of $\mathcal{M}$ in space s, and $\mathcal{I}$ contains no other structures, then none of the queries in Figure 4 matches. Hence $c \notin P^{\mathcal{I}}$.

Conversely, assume that c encodes a tree of $\mathcal{M}$ quasi-configurations in space s and $c \notin P^{\mathcal{I}}$. If none of the queries in Figure 4 (1) match, the head positions of every configuration must form a sequence $l, \ldots, l, h, r, \ldots, r$; hence all quasi-configurations are actually configurations. Queries (2)–(4) ensure that the first and last configuration are in the start and end state, respectively, and that each transition is matched by suitable state and tape modifications. Queries (5) ensure that tape cells that are not at the head of the TM are not modified between configurations. Queries (6) ensure that the movement of the head is consistent with the transitions, and especially does not leave the prescribed space. Note that the queries allow transitions that try to move the head beyond the tape and require that the head stays in its current position in this case. This allows the ATM to recognize the end of the tape, which is important for the Turing machines that we consider below. With all these restrictions observed, c must encode a run of $\mathcal{M}$ in space s. □

Proofs for Section 8

Lemma 14. *For any ATM $\mathcal{M}$, there is an* MDlog *query $P_1[x]$, a* LinMQ *$P_2[x]$, queries as in Definition 5 that are* LinMQ*s, and a same-cell query that is a* UCQ*, such that $P_1[x]$ and $P_2[x]$ containment-encode accepting runs of $\mathcal{M}$ in exponential space.*

Proof. Let $\mathcal{M} = \langle Q, \Sigma, \Delta, q_s, q_e\rangle$ with Q partitioned into existential states $Q_\exists$ and universal states $Q_\forall$. In order to use Lemma 12, we first construct queries P_1' and P_2' that containment-encode quasi-configuration trees of $\mathcal{M}$ in space 2^ℓ for some ℓ that is linear in the size of the queries (w.r.t. to suitable queries as in Definition 5).

Our signature contains the binary predicates (distinguished from the queries of Definition 5 by using lower case

letters) $\mathsf{firstConf}$, $\mathsf{nextConf}_\delta$ for all $\delta \in \Delta$, $\mathsf{firstCell}$, $\mathsf{nextCell}$, bit_i for all $i \in \{1, \ldots, \ell\}$, symbol, head, as well as the unary predicates $\mathsf{lastConf}$, and state_q for all $q \in Q$.

We define P'_1 to be the following MDlog query that has the goal predicate $\mathtt{U}_{goal}$ and uses two further constants 0 and 1:

$$\begin{aligned}
\mathsf{firstConf}(x,y) \wedge \mathtt{U}_{conf}(y) &\to \mathtt{U}_{goal}(x) && \\
\mathsf{state}_q(x) \wedge \mathsf{firstCell}(x,y) \wedge \mathtt{U}_{bit_1}(y) &\to \mathtt{U}_{conf}(x) && \text{for } q \in Q \\
\mathsf{bit}_{i-1}(x,0) \wedge \mathtt{U}_{bit_i}(x) &\to \mathtt{U}_{bit_{i-1}}(x) && \text{for } i \in \{2,\ldots,\ell\} \\
\mathsf{bit}_{i-1}(x,1) \wedge \mathtt{U}_{bit_i}(x) &\to \mathtt{U}_{bit_{i-1}}(x) && \text{for } i \in \{2,\ldots,\ell\} \\
\mathsf{symbol}(x,c_\sigma) \wedge \mathtt{U}_{symbol}(x) &\to \mathtt{U}_{bit_\ell}(x) && \text{for } \sigma \in \Sigma \\
\mathsf{head}(x,h) \wedge \mathtt{U}_{head}(x) &\to \mathtt{U}_{symbol}(x) && \\
\mathsf{head}(x,l) \wedge \mathtt{U}_{head}(x) &\to \mathtt{U}_{symbol}(x) && \\
\mathsf{head}(x,r) \wedge \mathtt{U}_{head}(x) &\to \mathtt{U}_{symbol}(x) && \\
\mathsf{nextCell}(x,y) \wedge \mathtt{U}_{bit_1}(y) &\to \mathtt{U}_{head}(x) && \\
\mathsf{nextConf}_\delta(x,y) \wedge \mathtt{U}_{conf}(y) &\to \mathtt{U}_{head}(x) && \text{for } \delta = \langle q,\sigma,q',\sigma',d\rangle \text{ with } q \in Q_\exists \\
\mathsf{nextConf}_{\delta_1}(x,y_1) \wedge \mathtt{U}_{conf}(y_1) \wedge & && \text{for } \delta_1 = \langle q,\sigma,q',\sigma',d\rangle, \\
\quad \mathsf{nextConf}_{\delta_2}(x,y_2) \wedge \mathtt{U}_{conf}(y_2) &\to \mathtt{U}_{head}(x) && q \in Q_\forall, \text{ and } \delta_1 \neq \delta_2 \\
\mathsf{lastConf}(x) &\to \mathtt{U}_{head}(x) &&
\end{aligned}$$

P'_1 encodes structures that resemble configuration trees, but with each configuration "tape" consisting of an arbitrary sequence of "cells" of the form $\mathsf{bit}_1(x,v_1), \ldots, \mathsf{bit}_\ell(x,v_\ell), \mathsf{symbol}(x,c_\sigma), \mathsf{head}(x,p)$, where each v_i is either 0 or 1. The values for the bit sequence encode a binary number of length ℓ. We provide a query P'_2 which ensures that each sequence of cells encodes an ascending sequence of binary numbers from $00\ldots0$ to $11\ldots1$. More precisely, P'_2 checks if there are any consecutive cells that violate this rule, i.e., the structures matched by P'_1 but not by P'_2 are those where each configuration contains 2^ℓ cells. The following query checks whether bit i is the rightmost bit containing a 0 and bit i in the successor configuration also contains a 0, which is a situation that must not occur if the bit sequences encode a binary counter:

$$\mathsf{bit}_i(y,0) \wedge \mathsf{bit}_{i+1}(y,1) \wedge \ldots \wedge \mathsf{bit}_\ell(y,1) \wedge \mathsf{nextCell}(y,z) \wedge \mathsf{bit}_i(z,0)$$

In a similar way, we can ensure that every bit to the right of the rightmost 0 is changed to 0, every bit that is left of a 0 remains unchanged, the first number is $0\ldots0$, and the last number is $1\ldots1$. The query P'_2 is the union of all of these (polynomially many) conditions, each with new atom $\mathsf{firstConf(x,y)}$ added and all variables other than x existentially quantified; this ensures that we obtain a unary query that matches the same elements as P'_1 if it matches at all.

We claim that the elements matching P'_1 but not P'_2 encode quasi-configuration trees of $\mathcal{M}$ in space 2^ℓ. Indeed, it is easy to specify the queries required by Definition 5. The most complicated query is $\mathsf{ConfCell}[x,y]$, which can be defined by the following LinMQ:

$$\begin{aligned}
\mathsf{state}_q(\lambda_1) \wedge \mathsf{nextCell}(\lambda_1,y) &\to \mathtt{U}(y) && \text{for all } q \in Q \\
\mathtt{U}(y) \wedge \mathsf{nextCell}(y,z) &\to \mathtt{U}(y) && \\
\mathtt{U}(\lambda_2) &\to \mathsf{hit} &&
\end{aligned}$$

The remaining queries are now easy to specify, where we use $\mathsf{ConfCell}[x,y]$, knowing that a conjunctive query over LinMQs can be transformed into a single LinMQ using Proposition 2:

$$\begin{aligned}
\mathsf{FirstConf}[x,y] &:= \mathsf{firstConf}(x,y) \\
\mathsf{NextConf}_\delta[x,y] &:= \exists z.\mathsf{ConfCell}(x,z) \wedge \mathsf{nextConf}_\delta(z,y) \\
\mathsf{LastConf}[x] &:= \exists z.\mathsf{ConfCell}(x,z) \wedge \mathsf{lastConf}(z) \\
\mathsf{State}_q[x] &:= \mathsf{state}_q(x) \\
\mathsf{Head}[x,y] &:= \mathsf{head}(x,y) \\
\mathsf{FirstCell}[x,y] &:= \mathsf{firstCell}(x,y)) \\
\mathsf{NextCell}[x,y] &:= \mathsf{nextCell}(x,y) \\
\mathsf{LastCell}[x] &:= \mathsf{lastConf}(x) \vee \exists z.\mathsf{nextConf}(x,z) \\
\mathsf{Symbol}[x,y] &:= \mathsf{symbol}(x,y) \\
\mathsf{SameCell}[x,y] &:= \exists v_1,\ldots,v_\ell.\mathsf{bit}_1(x,v_1) \wedge \mathsf{bit}_1(y,v_1) \wedge \\
&\qquad \ldots \wedge \mathsf{bit}_\ell(x,v_\ell) \wedge \mathsf{bit}_\ell(y,v_\ell)
\end{aligned}$$

Using these queries, we can construct a LinMQ P as in Lemma 12 such that $P_1 = P'_1$ and $P_2 = P'_2 \vee P$ containment-encode accepting runs of $\mathcal{M}$. □

Lemma 15. *Assume that there is some space bound s such that, for every DTM $\mathcal{M}$, there is a* MDlog *query $P_1[x]$ and an* MQ^{k+1} *query $P_2[x]$ with $k \geq 0$, such that $P_1[x]$ and $P_2[x]$ containment-encode accepting runs of $\mathcal{M}$ in s, where the queries required by Definition 5 are* MQ^{k+1} *queries. Moreover, assume that there is a suitable same-cell query that is in* MQ^k.

Then, for every ATM $\mathcal{M}'$, there is a MDlog *query $P'_1[x]$, an* MQ^{k+1} *$P'_2[x]$, and* MQ^{k+1} *queries as in Definition 5, such that $P'_1[x]$ and $P'_2[x]$ containment-encode an accepting run of $\mathcal{M}'$ in space $s' \geq 2^s$. Moreover, the size of the queries for this encoding is polynomial in the size of the queries for the original encoding.*

Proof. There is a TM $\mathcal{M} = \langle Q, \Sigma, \Delta, q_s, q_e\rangle$ that counts from 0 to 2^s in binary (using space s) and then halts. $\mathcal{M}$ can be small (constant size) since our formalization of (A)TMs allows the TMs to recognize the last tape position to ensure that the maximal available space is used. The computation will necessarily take $s' > 2^s$ steps to complete since multiple steps are needed to increment the counter by 1. Let $P_1[x]$ and $P_2[x]$ be queries that containment-encode accepting runs of $\mathcal{M}$ in s, and let $\mathsf{ConfCell}$, $\mathsf{SameCell}$, etc. denote the respective LinMQ^k as in Definition 5.

Let $\mathcal{M}' = \langle Q', \Sigma', \Delta', q'_s, q'_e\rangle$ be an arbitrary ATM. We use the signature of P_1, extended by additional binary predicates $\mathsf{firstConf}'$, $\mathsf{nextConf}'_\delta$ for all $\delta \in \Delta'$, symbol', head', as well as unary predicates $\mathsf{lastConf}'$, and state'_q for all $q \in Q'$. All of these are assumed to be distinct from predicates in P_1.

Let $\mathtt{U}_{\mathrm{goal}}$ be the goal predicate of P_1, and let $\mathtt{U}_{\mathrm{tape}}$ be a new unary IDB predicate. We construct the program $\bar{P}_1$ from P_1 as follows. For every rule of P_1 that does not contain an IDB atom in its body we add the atom $\mathtt{U}_{\mathrm{tape}}(x)$ to the body, where x is any variable that occurs in the rule. Intuitively speaking, the IDBs $\mathtt{U}_{\mathrm{tape}}$ and $\mathtt{U}_{\mathrm{goal}}$ mark the start and end of tapes of $\mathcal{M}'$, which are represented by runs of $\mathcal{M}$. Moreover, we

modify $\bar{P}_1$ to "inject" additional state and head information for $\mathcal{M}'$ into configurations of $\mathcal{M}$, i.e., we extend P_1 to ensure that every element e with $\mathsf{state}_q(e)$ also occurs in some $\mathsf{symbol}'(e, c'_{\sigma'})$ and in some relation $\mathsf{head}'(e, p)$. This can always be achieved by adding a linear number of IDB predicates and rules.

Now P'_1 is defined to be a MDlog query with goal predicate $\mathtt{U}'_{\text{goal}}$ (assumed, like all IDB predicates of form $\mathtt{U}'$ below, to be distinct from any IDB predicate in $\bar{P}_1$), which is obtained as the union of $\bar{P}_1$ with the following rules:

$$\begin{aligned}
\mathsf{firstConf}'(x,y) \wedge \mathtt{U}'_{conf}(y) &\to \mathtt{U}'_{goal}(x) && \\
\mathsf{state}'_q(x) \wedge \mathtt{U}_{goal}(x) &\to \mathtt{U}'_{conf}(x) && \text{for } q \in Q \\
\mathsf{nextCell}'(x,y) \wedge \mathtt{U}_{goal}(y) &\to \mathtt{U}_{tape}(x) && \text{for } q \in Q \\
\mathsf{nextConf}'_\delta(x,y) \wedge \mathtt{U}'_{conf}(y) &\to \mathtt{U}_{tape}(x) && \text{for } \delta = \langle q, \sigma, q', \sigma', d\rangle \\
& && \text{with } q \in Q_\exists \\
\mathsf{nextConf}'_{\delta_1}(x,y_1) \wedge \mathtt{U}'_{conf}(y_1) \wedge & && \text{for } \delta_1 = \langle q, \sigma, q', \sigma', d\rangle, \\
\mathsf{nextConf}'_{\delta_2}(x,y_2) \wedge \mathtt{U}'_{conf}(y_2) &\to \mathtt{U}_{tape}(x) && q \in Q_\forall, \text{ and } \delta_1 \neq \delta_2 \\
\mathsf{lastConf}'(x) &\to \mathtt{U}_{tape}(x) &&
\end{aligned}$$

P'_1 encodes trees of trees of $\mathcal{M}$ quasi-configurations in space s. The structures matched by P'_1 but not by P_2 encode trees of accepting runs of $\mathcal{M}$ in space s (note that these runs are linear, since $\mathcal{M}$ is not alternating). Every such run consists of the same number $s' \geq 2^s$ of configurations; these configurations represent the tape cells of our encoding of $\mathcal{M}'$ sequences. This encoding is formalized by queries as follows. The queries $\mathsf{FirstConf}'[x,y]$, $\mathsf{State}'_q[x]$, $\mathsf{Head}'[x,y]$, and $\mathsf{Symbol}'[x,y]$ are directly expressed by singleton CQs that use the eponymous predicates $\mathsf{firstConf}'(x,y)$, etc. To access cells of $\mathcal{M}'$, we can use the analogous queries to access configurations of $\mathcal{M}$: $\mathsf{FirstCell}'[x,y] = \mathsf{FirstConf}(x,y)$, $\mathsf{NextCell}'[x,y] = \mathsf{NextConf}(x,y)$, and $\mathsf{LastCell}'[x] = \mathsf{LastConf}(x)$.

The remaining queries can be expressed as LinMQ queries. To present these queries in a more readable way, we specify them in regular expression syntax rather than giving many rules for each. It is clear that regular expressions over unary and binary predicates can be expressed in LinMQ (it was already shown that MQs can express regular path queries, which is closely related [19]). We use abbreviation P1SYMBOL to express the regular expression that is a disjunction of all predicate symbols that occur in P_1 (this allows us to skip over any structures generated by P_1; with the specific forms of P_1 that can occur in our proofs, one could make this more specific to use only certain binary predicates, but our formulation does not depend on internals of P_1). Moreover, let STATE be the disjunction of all atoms $\mathsf{state}'_q(x)$ and $\exists y.\mathsf{head}'(x,y)$ (both unary).

$$\begin{aligned}
\mathsf{NextConf}'_\delta[x,y] &\coloneqq \mathsf{STATE\ P1SYMBOL}^*\ \mathsf{nextConf}'_\delta \\
\mathsf{LastConf}'[x] &\coloneqq \mathsf{STATE\ P1SYMBOL}^*\ \mathsf{lastConf}' \\
\mathsf{ConfCell}'[x,y] &\coloneqq \mathsf{STATE\ P1SYMBOL}^*\ \mathsf{HEAD}
\end{aligned}$$

The unary query $\mathsf{LastConf}'[x]$ uses the variable at the beginning of the expression as its answer. It is easy to verify that the elements accepted by P'_1 but not by P_2 encode sequences of quasi-configurations of $\mathcal{M}'$ in space s' with respect to these queries. To apply Lemma 12, we need to specify an additional $\mathsf{SameCell}'$ query for this encoding.

$\mathsf{SameCell}'$ is expressed by an MQ^{k+1} query that can in general not be expressed by a MQ^k query:

$$\begin{aligned}
\mathsf{FirstCell}(\lambda_1, x) &\to \mathtt{U}_1(x) \\
\mathtt{U}_1(x) \wedge \mathsf{NextCell}(x,x') &\to \mathtt{U}_1(x') \\
\mathsf{State}_q(\lambda_1) \wedge \mathsf{FirstCell}(\lambda_1,x) \wedge \mathsf{Symbol}(x,z) \wedge \mathsf{Head}(x,v) \wedge & \\
\quad \mathsf{State}_q(\lambda_2) \wedge \mathsf{FirstCell}(\lambda_2,y) \wedge \mathsf{Symbol}(y,z) \wedge \mathsf{Head}(y,v) &\to \mathtt{U}_2(y) \\
& \text{for all } q \in Q \\
\mathtt{U}_1(x) \wedge \mathtt{U}_2(y) \wedge \mathsf{SameCell}(x,y) \wedge & \\
\mathsf{NextCell}(x,x') \wedge \mathsf{Symbol}(x',z) \wedge \mathsf{Head}(x',v) \wedge & \\
\mathsf{NextCell}(y,y') \wedge \mathsf{Symbol}(y',z) \wedge \mathsf{Head}(y',v) &\to \mathtt{U}_2(y') \\
\mathtt{U}_2(y) \wedge \mathsf{LastCell}(y) &\to \mathsf{hit}
\end{aligned}$$

where FirstCell, Symbol, SameCell, and LastCell are the queries for which P_1 and P_2 containment-encode runs of $\mathcal{M}$. Note that our constructions already ensure that the sequences of $\mathcal{M}$-cells compared by $\mathsf{SameCell}'$ are of the same length.

To complete the proof, we apply Lemma 12 to construct an MQ^{k+1} $\bar{P}_2$. The MQ^{k+1} P'_2 is obtained by expressing the disjunction of P_2 and $\bar{P}_2$ as an MQ^{k+1} using Proposition 2. Then P'_1 and P'_2 containment encode accepting runs of $\mathcal{M}'$ in space s'. □

Theorem 13. *Deciding containment of* MDlog *queries in* MQ^k *queries is hard for* $(k+2)$ExpTime.

Proof. The claim is shown by induction on k. For the base case, we show that deciding containment of MQ queries is 3ExpTime-hard. By Lemma 14, for any DTM $\mathcal{M}^0$, there is a MDlog query P^0_1, a LinMQ P^0_2, LinMQs as in Definition 5, and a same-cell query that is a UCQ with respect to which P^0_1 and P^0_2 containment-encode accepting runs of $\mathcal{M}^0$ in exponential space s. By applying Lemma 15, we obtain, for an arbitrary ATM $\mathcal{M}^1$, a MDlog query P^1_1, an MQ P^1_2, and MQ queries as in Definition 5 (including a same-cell query), that containment-encode accepting runs of $\mathcal{M}^1$ in space $s' \geq 2^s$.

The induction step for $k > 1$ is immediate from Lemma 15. □