View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

Datalog with Negation and Monotonicity

Bas Ketsman[1]
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Christoph Koch
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
christoph.koch@epfl.ch

Abstract

Positive Datalog has several nice properties that are lost when the language is extended with negation. One example is that fixpoints of positive Datalog programs are robust w.r.t. the order in which facts are inserted, which facilitates efficient evaluation of such programs in distributed environments. A natural question to ask, given a (stratified) Datalog program with negation, is whether an equivalent positive Datalog program exists.

In this context, it is known that positive Datalog can express only a strict subset of the monotone queries, yet the exact relationship between the positive and monotone fragments of semi-positive and stratified Datalog was previously left open. In this paper, we complete the picture by showing that monotone queries expressible in semi-positive Datalog exist which are not expressible in positive Datalog. To provide additional insight into this gap, we also characterize a large class of semi-positive Datalog programs for which the dichotomy 'monotone if and only if rewritable to positive Datalog' holds. Finally, we give best-effort techniques to reduce the amount of negation that is exhibited by a program, even if the program is not monotone.

2012 ACM Subject Classification Information systems → Relational database query languages; Theory of computation → Constraint and logic programming

Keywords and phrases Datalog, Monotonicity

Digital Object Identifier 10.4230/LIPIcs.ICDT.2020.19

1 Introduction

Within classical model theory, several results exist that equate syntactic with semantic restrictions of first-order logic (FOL). One example is the homomorphism preservation theorem [22], which states that the fragment of FOL that is preserved under homomorphisms has the same expressive power as the set of existential positive FOL formulas. Analogously, Lyndon's preservation theorem states that the fragment preserved under surjective homomorphisms equals the set of positive FOL formulas. (Recall that query q is preserved under homomorphisms if, for all instances I, J and mappings h, $h(I) \subseteq J$ implies $h(q(I)) \subseteq q(J)$.)

For finite structures, which are the central interest in database theory, it is well-known that most of such equalities fail. For example, for Lyndon's theorem this was shown in the 80's by Ajtai and Gurevich [4] (and by Stolboushkin [25], using a simplified counterexample). One of the only exceptions is the homomorphism theorem, which Rossman [22] proved to hold in the finite as well. A similar result exists in the context of Datalog. Here, Feder and Vardi [12] showed that the fragment of semi-positive Datalog (in which negation is allowed only over extensional atoms) preserved under homomorphisms has the same expressive power as the positive fragment of Datalog. That the latter result does not transfer to general fixpoint logics was shown by Dawar and Kreutzer [10].

[1] Currently at Vrije Universiteit Brussel, Brussels, Belgium

© Bas Ketsman and Christoph Koch;
licensed under Creative Commons License CC-BY
23rd International Conference on Database Theory (ICDT 2020).
Editors: Carsten Lutz and Jean Christoph Jung; Article No. 19; pp. 19:1–19:18

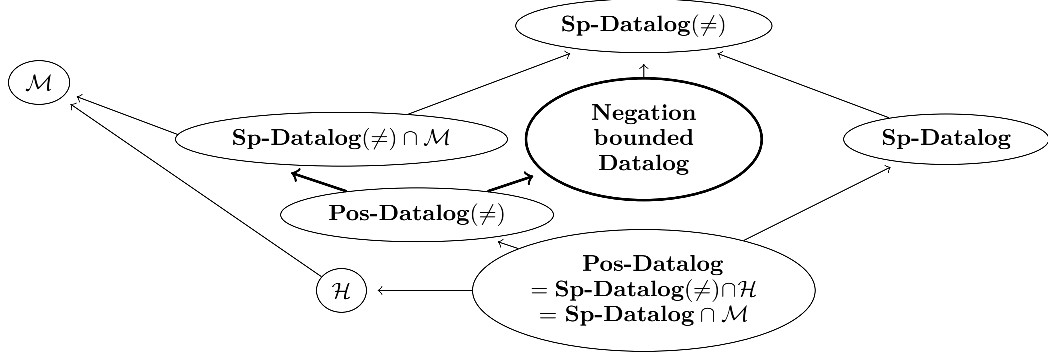


Figure 1 Schematic overview of the different fragments of semi-positive Datalog that we consider and the relationship between their expressive power and the classes of homomorphically closed queries (denoted $\mathcal{H}$) and monotone queries (denoted $\mathcal{M}$). Here, arrows mean subsumption.

In this paper, we study the relationship between the positive and the monotone fragment of stratified Datalog (which allows negation in a stratified fashion) and semi-positive Datalog (which allows negation over existential predicates only; thus consisting of the single stratum programs). Their positive fragment is the fragment in which all forms of negation are forbidden. Their monotone fragment is the subclass of programs expressing a monotone query q; thus with $I \subseteq J$ implying $q(I) \subseteq q(J)$ for all instances I and J. It is known that positive Datalog can only express monotone queries [3, 5, 18], and that some polynomial time computable monotone queries are not expressible in positive Datalog [3, 10]. To the best of our knowledge it remained open whether such queries exist that are themselves expressible in stratified Datalog.

Our first set of contributions addresses this gap and proves the relationships that were previously left open:

(a) The monotone fragment of stratified Datalog without inequalities is strictly more expressive than positive Datalog, even when restricted to two-stratum programs. (Theorem 7.2)

(b) The monotone fragment of stratified Datalog with inequalities is strictly more expressive than positive Datalog, even when restricted to single-stratum programs. (Theorem 4.1)

(c) The monotone fragment of semi-positive Datalog without inequalities is equally expressive as positive Datalog without inequalities. (Theorem 5.1)

Motivated by contributions (a) and (b), we explore further the expressivity gap between the monotone fragment of semi-positive Datalog and positive Datalog:

(d) Based on the notion conflict-freedom, we identify a large fragment of semi-positive Datalog, called negation-bounded Datalog, for which the two restrictions coincide. (Theorem 6.2) We show that deciding conflict freedom is EXP-complete, respectively, *co*NP-complete if a bound is assumed on the arities of relations. (Theorem 3.13)

Our motivation to study the monotone fragment and the positive fragment of Datalog with negation is driven by an underlying interest in the declarative networking paradigm [1, 19], which is concerned with the design of network programs in extensions of Datalog. In this setting, it is folklore knowledge that positive Datalog programs can be computed efficiently via asynchronous pipelined joins [19, 16], because Datalog rules without negation can fire independently of each other, without a need for synchronisation. This is in stark contrast to the case where rules have negated atoms, as then a round of consensus is needed to reach

agreement on the absence of facts. A model-theoretic explanation for this observation is that fixpoints of positive Datalog programs are robust w.r.t. the order in which facts are inserted. Monotonicity on the other hand is recognized as the facilitating property for this observation and as theoretical upper-bound on what can be computed efficiently [6, 16, 17, 28]. Not surprisingly, the terms positive and monotone are often mentioned in the same breath.

Our contributions (a-b) shows that the terms positive and monotone do not always interchange, even in the context of a language as simple as semi-positive Datalog.

We note that avoiding negation completely is usually not desirable, as this puts a significant limitation on the type of programs that can be formulated. Nevertheless, it is common in practice to strive for both goals: To find an optimal rewrite for a program (*i.e.*, equivalent with less exposure to negation) as well as to give up robustness in favour of computational properties (*i.e.*, less exposure to negation at the cost of some expressive power). A real-world example of the latter is the choice of the 2-phase commit (2PC) protocol to support atomicity of distributed transactions: While it is well-known that 2PC blocks under certain types of failures, it is usually favoured over more robust alternatives, because their robustness comes at the cost of additional rounds of consensus, and thus higher latency [14, 15]. When formulated in a logical language, consensus is recognisable as universal quantification, which in Datalog-like languages is encoded through negation.

In non-distributed contexts a similar trade-off exists, which is in terms of the number of strata that the program admits. Indeed, while there are several well-known optimization techniques to evaluate single-stratum programs, like semi-naive evaluation and magic-set optimization, traditional Datalog engines evaluate the strata of a stratified Datalog program one after another and thus benefit from techniques that reduce the number of strata.

Our final contribution elaborates on this by addressing the negation elimination problem:

(e) We describe how the exposure to negation in stratified Datalog programs can be reduced even if the program is not monotone. Together, these techniques form a best effort procedure to remove negated atoms from programs, which we show to run with exponential space (respectively polynomial space if a bound on the arity of relations is assumed). (Theorem 8.5) Given that almost all properties for Datalog are undecidable, this is essentially the best one can hope for.

Outline

In Section 2, we give the essential definitions that are used throughout the paper. Section 3 covers the concept of conflict-free proof trees, which is central in several of our results. In Section 4, we show that the monotone fragment of semi-positive Datalog is not expressible in positive Datalog. In Section 5 and Section 6, we give positive results on this equality for when no inequalities occur in the considered programs, or when a bound exists on the number of negated facts that can occur in proof trees of a program. Finally, in Section 7, we describe best-effort techniques that can be used to remove negation from programs independent of whether these programs are monotone.

2 Preliminaries

In this section, we give an overview of the necessary concepts and definitions that are used throughout the paper.

For positive integers n, henceforth we abbreviate the set $\{1, \ldots, n\}$ by $[n]$.

2.1 Schemas and Instances

As usual, a *(database) schema* σ is a set of relation names R with associated arities $\mathit{arity}(R)$. We often write $R^{(r)} \in \sigma$, as abbreviation for $R \in \sigma$ with $\mathit{arity}(R) = r$.

Throughout the paper, we assume the existence of an infinite domain **dom** of data values. Given schema σ, a *fact* $\boldsymbol{f}$ over σ is then defined as a tuple $\boldsymbol{f} := R(\bar{t})$, with $R^{(r)} \in \sigma$ and $\bar{t} \in \mathbf{dom}^r$. By $\mathit{Facts}(\sigma)$ we denote the infinite set consisting of all facts over σ. A *(database) instance* I over σ is a finite subset of $\mathit{Facts}(\sigma)$.

2.2 Queries

A *query* q is a mapping from instances over some database schema σ_1 to instances over another database schema σ_2. We call σ_1 the *input schema* and σ_2 the *output schema* of q. As usual, we assume queries to be *generic*, which means that $\pi(q(I)) = q(\pi(I))$, for every permutation π of **dom**.

For two instances I and J over σ_1, we call a mapping h a *homomorphism from I to J*, if $h(I) \subseteq J$. We say that query q is *preserved under homomorphisms* (also called *strongly monotone* [3]) if for every homomorphism h from some instance I to instance J, $h(q(I)) \subseteq q(J)$. We say that q is *monotone* if $I \subseteq J$ implies $q(I) \subseteq q(J)$, for every pair of instances I and J, Henceforth, we denote the class of all queries that are preserved under homomorphisms by $\mathcal{H}$ and the class of all monotone queries by $\mathcal{M}$.

A query q is *contained* in a query q', denoted $q \subseteq q'$, if $q(I) \subseteq q'(I)$ for every instance I.

2.3 Datalog with Negation

We define semi-positive Datalog. For this, let **var** be an infinite domain of variables, disjoint from **dom**, and let σ be a schema. An *atom $R(\overline{x})$ over σ* consists of a relation name $R^{(r)} \in \sigma$ and a tuple $\overline{x}$ from $\mathbf{var}^r$. We do not allow constants in atoms. For a set U of atoms, we write $\mathit{Vars}(U)$ to denote the set of all variables used by the atoms in U.

A *Datalog rule* τ over σ has the following form:

$$H(\overline{y}) \leftarrow R_1(\overline{x_1}), \ldots, R_\ell(\overline{x_k}), \neg S_1(\overline{z_1}), \ldots, \neg S_m(\overline{z_m}), \beta_1, \ldots, \beta_n.$$

Here, for every $i \in [\ell]$ and $j \in [m]$, $H(\overline{y})$, $R_i(\overline{x_i})$ and $S_j(\overline{z_j})$ are atoms over σ, and, for every $k \in [n]$, β_i is an inequality of the form $x \neq y$, with $\{x, y\} \subseteq \mathbf{var}$. Henceforth, we also refer to $H(\overline{y})$ by head_τ (the *head* of τ); to $\{R_i(\overline{x_i}) \mid i \in [\ell]\}$ by Pos_τ (the *positive body atoms in* τ); to $\{S_i(\overline{z_i}) \mid i \in [m]\}$ by Neg_τ (the *negated body atoms in* τ); and finally to $\{\beta_1, \ldots, \beta_n\}$ by Ineq_τ (the *inequalities in* τ). As usual, we only consider safe rules, thus with $\mathit{Vars}(\mathit{Neg}_\tau \cup \mathit{Ineq}_\tau \cup \{\mathit{head}_\tau\}) \subseteq \mathit{Vars}(\mathit{Pos}_\tau)$.

For a schema σ, a *Semi-positive Datalog program P over σ* is a set of Datalog rules P. As usual, we call relation names from σ that occur in the head of a rule in P *intensional* and all others *extensional.* For rules τ in P, the set Neg_τ must contain only atoms with extensional relation names. By *Sp-Datalog*($\neq$) we denote the class of all semi-positive Datalog programs. We also consider the following subclasses: *Sp-Datalog* (semi-positive Datalog without inequalities) denotes the programs in which Ineq_τ is empty for every rule τ; *Pos-Datalog*($\neq$) (positive Datalog) denotes the programs in which Neg_τ is empty for every rule τ; and *Pos-Datalog* (Positive Datalog without inequalities) denotes the intersection of the latter two, thus in which Ineq_τ and Neg_τ are empty for every rule τ.

Since we are interested in Datalog programs that express queries, we assume that the schema σ that a Datalog program P is defined over has distinguished input and output

relation names. We denote these by $in(P) \subseteq \sigma$ and $out(P) \subseteq \sigma$, respectively. When not explicitly mentioned, we assume that $in(P)$ coincides with the extensional relation names in P and that $out(P) = \{\mathsf{Output}^{(k)}\}$, for some integer $k \geq 0$.

▶ **Example 2.1.** As a running example throughout this paper, we consider semi-positive Datalog program P_Δ. This program is defined over schema $\sigma := \{\mathsf{Edge}^{(2)}, \mathsf{T1}^{(2)}, \mathsf{T2}^{(2)}, \mathsf{Output}^{(2)}\}$, with $in(P_\Delta) = \{\mathsf{Edge}\}$ and $out(P_\Delta) = \{\mathsf{Output}\}$, and has the following rules:

$$\mathsf{T1}(\mathsf{x}, \mathsf{y}) \leftarrow \mathsf{Edge}(\mathsf{x}, \mathsf{y}), \mathsf{Edge}(\mathsf{y}, \mathsf{z}), \neg\mathsf{Edge}(\mathsf{z}, \mathsf{x}). \tag{1}$$

$$\mathsf{T2}(\mathsf{x}, \mathsf{y}) \leftarrow \mathsf{Edge}(\mathsf{x}, \mathsf{y}), \neg\mathsf{Edge}(\mathsf{y}, \mathsf{z}), \mathsf{Edge}(\mathsf{z}, \mathsf{x}). \tag{2}$$

$$\mathsf{Output}(\mathsf{x}, \mathsf{y}) \leftarrow \mathsf{Edge}(\mathsf{x}, \mathsf{y}), \mathsf{Edge}(\mathsf{y}, \mathsf{z}), \mathsf{Edge}(\mathsf{z}, \mathsf{x}). \tag{3}$$

$$\mathsf{Output}(\mathsf{x}, \mathsf{y}) \leftarrow \mathsf{T1}(\mathsf{x}, \mathsf{y}), \mathsf{T2}(\mathsf{x}, \mathsf{y}), \mathsf{x} \neq \mathsf{y}. \tag{4}$$

Intuitively, P_Δ expects a directed graph as input and asks for edges (a, b) for which one of the following properties is true: (a, b) is part of a triangle; or (a, b) is part of two open triangles, one in which the edge to a is missing, the other in which the edge from b is missing. We will later show (in Proposition 3.12) that P_Δ is of particular interest because it expresses a monotone query.

2.4 Proof Tree Semantics

The semantics of Datalog is usually defined bottom-up, in terms of an immediate consequence operator. We use the equivalent top-down definition via proof trees. (We refer to [2] for a detailed discussion on their equivalence.) For a formal definition, we first define a *prevaluation v for a Datalog rule τ* as a mapping from the variables occurring in τ to values from **dom**. A prevaluation v for τ is a *valuation for τ* if each inequality $x \neq y \in Ineq_\tau$ admits $v(x) \neq v(y)$. Before defining the concept proof tree, we first define a slightly weaker concept, which we call a candidate proof tree:

▶ **Definition 2.2.** *Given a fact $\boldsymbol{f}$, instance I and program $P \in$ Sp-Datalog($\neq$), a* candidate proof tree $\mathcal{T}$ of $\boldsymbol{f}$ from I and P *is a labeled tree with the following properties:*

1. *Each vertex is labeled with a fact;*
2. *Each leaf is labeled with a fact $\boldsymbol{g}$ over an extensional relation name, and with either sign '+', if $\boldsymbol{g} \in I$, or '-', if $\boldsymbol{g} \notin I$.*
3. *The root is labeled with $\boldsymbol{f}$;*
4. *Each intermediary vertex is associated with a rule $\tau \in P$ and prevaluation v for τ, such that its label equals $v(head_\tau)$, and for each atom $A \in Body_\tau$ it has a child whose label equals $v(A)$. If A has an extensional relation name this child must have a sign that equals '+' if $A \in Pos_\tau$ and '-' if $A \in Neg_\tau$.*

Unless stated otherwise, we assume throughout the paper that the root of a candidate proof tree is always labeled with a fact from an output relation. Further, we denote by $Fringe^+_\mathcal{T}$ the set of all extensional facts that occur as labels for leafs in $\mathcal{T}$ with sign '+', and by $Fringe^-_\mathcal{T}$ the set of extensional facts for leafs with sign '$-$'.

▶ **Definition 2.3.** *A* proof tree $\mathcal{T}$ of $\boldsymbol{f}$ from I and $P \in$ *Sp-Datalog($\neq$) is a candidate proof tree of $\boldsymbol{f}$ from I and P in which all prevaluations are valuations (for the respective rule), $Fringe^+_\mathcal{T} \subseteq I$, and $Fringe^-_\mathcal{T} \cap I = \emptyset$.*

We sometimes refer to a (candidate) proof tree $\mathcal{T}$ from program P without specifying a fact or instance. In that case, we assume that $\mathcal{T}$ is a (candidate) proof tree of the fact that its root is labeled with, and that it is a proof tree from *some* instance I and P.

Every semi-positive Datalog program P expresses a unique query $q_P : \mathit{in}(P) \mapsto \mathit{out}(P)$ that is defined by P and its distinguished input and output relation names. Its evaluation over instances $I \subseteq \mathit{Facts}(\mathit{in}(P))$, denoted by $q_P(I)$, is defined by the set of all facts $\boldsymbol{f} \in \mathit{Facts}(\mathit{out}(P))$ for which a proof tree of $\boldsymbol{f}$ from I and P exists. Henceforth, we say that two programs P_1 and P_2 having the same input and output relation names are *equivalent* if they express the same query, thus with $q_{P_1}(I) = q_{P_2}(I)$ for every instance $I \subseteq \mathit{Facts}(\mathit{in}(P_1))$. For every class of Datalog programs we consider also the class of queries that are expressed by these programs. To distinguish between the two, the latter are always in boldface. For example, **Sp-Datalog**($\neq$) refers to the class of queries expressible with *Sp-Datalog*($\neq$) programs.

3 Conflicts

In this section, we introduce the main machinery that we use to reason about the relationship between proof trees from different programs. We start with two simple constructions. Given a semi-positive Datalog program P, P^+ denotes the program in *Pos-Datalog*($\neq$) obtained by removing all negated atoms from rules in P; P^* denotes the program in *Pos-Datalog* obtained by removing, in addition to the negated atoms, also all inequalities from rules in P^+. We leave the schema definitions untouched, thus $\mathit{in}(P) = \mathit{in}(P^+) = \mathit{in}(P^*)$, and $\mathit{out}(P) = \mathit{out}(P^+) = \mathit{out}(P^*)$.

▶ **Example 3.1.** For an example of the constructions, take program P_Δ from Example 2.1. Then, P_Δ^+ contains the following rules:

$$\begin{aligned}
\mathsf{T1}(\mathsf{x},\mathsf{y}) &\leftarrow \mathsf{Edge}(\mathsf{x},\mathsf{y}), \mathsf{Edge}(\mathsf{y},\mathsf{z}).\\
\mathsf{T2}(\mathsf{x},\mathsf{y}) &\leftarrow \mathsf{Edge}(\mathsf{x},\mathsf{y}), \mathsf{Edge}(\mathsf{z},\mathsf{x}).\\
\mathsf{Output}(\mathsf{x},\mathsf{y}) &\leftarrow \mathsf{Edge}(\mathsf{x},\mathsf{y}), \mathsf{Edge}(\mathsf{y},\mathsf{z}), \mathsf{Edge}(\mathsf{z},\mathsf{x}).\\
\mathsf{Output}(\mathsf{x},\mathsf{y}) &\leftarrow \mathsf{T1}(\mathsf{x},\mathsf{y}), \mathsf{T2}(\mathsf{x},\mathsf{y}), \mathsf{x} \neq \mathsf{y}.
\end{aligned}$$

Program P_Δ^* has the next rules:

$$\begin{aligned}
\mathsf{T1}(\mathsf{x},\mathsf{y}) &\leftarrow \mathsf{Edge}(\mathsf{x},\mathsf{y}), \mathsf{Edge}(\mathsf{y},\mathsf{z}).\\
\mathsf{T2}(\mathsf{x},\mathsf{y}) &\leftarrow \mathsf{Edge}(\mathsf{x},\mathsf{y}), \mathsf{Edge}(\mathsf{z},\mathsf{x}).\\
\mathsf{Output}(\mathsf{x},\mathsf{y}) &\leftarrow \mathsf{Edge}(\mathsf{x},\mathsf{y}), \mathsf{Edge}(\mathsf{y},\mathsf{z}), \mathsf{Edge}(\mathsf{z},\mathsf{x}).\\
\mathsf{Output}(\mathsf{x},\mathsf{y}) &\leftarrow \mathsf{T1}(\mathsf{x},\mathsf{y}), \mathsf{T2}(\mathsf{x},\mathsf{y}).
\end{aligned}$$

Both constructions result in well-defined Datalog programs. Particularly notice that the constructions preserve safeness of the programs by only removing negated atoms and inequalities, whose variables all occur also in non-negated atoms (due to safeness of the original program P).

We conclude this section with the following observation:

▶ **Proposition 3.2.** *Let σ be a database schema and P a semi-positive Datalog program over σ. Then, $q_P \subseteq q_{P^+} \subseteq q_{P^*}$.*

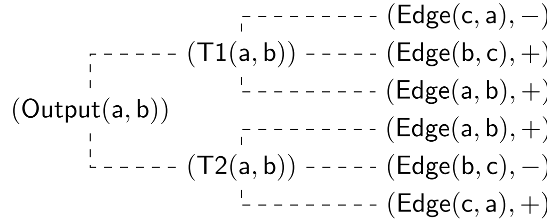
3.1 Fringe and Inequality Conflicts

To reason about the other direction of the containments in Proposition 3.2 (that is, $q_{P^*} \subseteq q_P$ and $q_{P^+} \subseteq q_P$), we need to reason about subtle differences in the proof trees that these programs admit. For this purpose, our distinction between proof trees and candidate proof

trees comes in handy. First, recall that every proof tree $\mathcal{T}$ from an instance I and program $P \in$ *Sp-Datalog*($\neq$) is a candidate proof tree from I and P (by definition). The opposite direction is not true, because a candidate proof tree $\mathcal{T}$ may admit

- *fringe conflicts*, a term we use to refer to facts in $\mathit{Fringe}^+_{\mathcal{T}} \cap \mathit{Fringe}^-_{\mathcal{T}}$; and
- *inequality conflicts*, a term we use to refer to inequalities in rules (associated to vertices of $\mathcal{T}$) that are not made true by the associated prevaluation.

▶ **Example 3.3.** For an example of a proof tree with fringe conflicts, consider the proof tree for program $P_\triangle$ from Example 2.1 that is given below.



The following relationship applies:

▶ **Proposition 3.4.** *A candidate proof tree $\mathcal{T}$ from semi-positive Datalog program P is a proof tree from P if and only if $\mathcal{T}$ is free of fringe and inequality conflicts.*

3.2 Expansion Trees

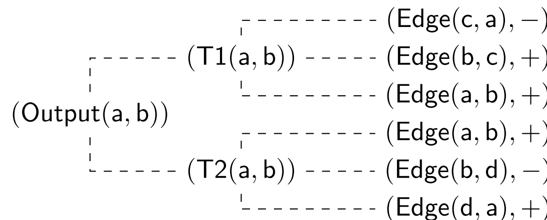
A special type of candidate proof tree is the (unfolding) expansion tree [9], which we generalize here for semi-positive Datalog:

▶ **Definition 3.5.** *An* expansion tree *is a candidate proof tree $\mathcal{T}$ in which every intermediate vertex n (including the root) is associated with a rule τ and prevaluation V, such that V maps every pair of different variables not occurring in head_τ onto different values that all occur only in the subtree of $\mathcal{T}$ with n as root.*

While a candidate proof tree for a semi-positive Datalog program is not necessarily an expansion tree, it always is the homomorphic image of an expansion tree. Henceforth we use the following naming conventions: We denote expansion trees by $\mathcal{T}_e$. For a mapping g over **dom** and candidate proof tree $\mathcal{T}$, $g(\mathcal{T})$ denotes the candidate tree obtained by substituting all facts $\boldsymbol{f}$ occurring as labels in $\mathcal{T}$ and all valuations v associated to vertices in $\mathcal{T}$ by their respective images $g(\boldsymbol{f})$ and $g \circ v$ under g.

▶ **Proposition 3.6.** *Let $P \in$ Sp-Datalog($\neq$). For every candidate proof tree $\mathcal{T}$ from P, there is an expansion tree $\mathcal{T}_e$ and mapping g such that $g(\mathcal{T}_e) = \mathcal{T}$. Moreover, if $\mathcal{T}$ is free of fringe and inequality conflicts, then $\mathcal{T}_e$ is free of fringe and inequality conflicts.*

▶ **Example 3.7.** The below tree is an example expansion tree for $P_\triangle$.



3.3 Conflicts and Monotonicity

In the remainder of this section, we show the relevance of conflicts in the relationship between monotone and positive programs. First, we distinguish two categories of programs depending on which conflicts their candidate proof trees admit.

▶ **Definition 3.8.** *Let $P \in$ Sp-Datalog($\neq$). Program P is* conflict-free *if each candidate proof tree from P without inequality conflicts is without fringe conflicts. Program P is* free of explicit conflicts *if each expansion tree from P is without fringe and inequality conflicts.*

The intuition behind the term explicit conflict is that expansion trees are less diverged from the rules of the program than arbitrary candidate proof trees are, and that conflicts in expansion trees therefore can be observed more easily than fringe conflicts by looking at the wiring of variables throughout rules in the program.

▶ **Proposition 3.9.** *For $P \in$ Sp-Datalog($\neq$) we have the following equivalences:*
1. *$q_P = q_{P^*}$ if $q_P \in \mathcal{H}$ and P is free of explicit conflicts;*
2. *$q_P = q_{P^+}$ if $q_P \in \mathcal{M}$ and P is conflict-free.*

Proof. Since $q_P \subseteq q_{P^*}$ and $q_P \subseteq q_{P^+}$ follow from Proposition 3.2, we need to show only $q_{P^*} \subseteq q_P$ and $q_{P^+} \subseteq q_P$.

(1) Let I be an arbitrary instance and $\boldsymbol{f}$ an arbitrary fact in $q_{P^*}(I)$. Let $\mathcal{T}^*$ denote the proof tree of $\boldsymbol{f}$ from I and P^*. We show $\boldsymbol{f} \in q_P(I)$.

By Proposition 3.6, there exists an expansion tree $\mathcal{T}_e^*$ for P^* that is without fringe and inequality conflicts and a mapping g, with $g(\mathcal{T}_e^*) = \mathcal{T}^*$. It follows from the construction of P^* that $\mathcal{T}_e^*$ can be extended to an expansion tree $\mathcal{T}_e$ from P: For vertices n, let τ_n^* and V_n denote its associated rule and valuation. Then let $\mathcal{T}_e$ be the candidate proof tree obtained from $\mathcal{T}_e^*$ by replacing, for each vertex n, its rule τ_n^* by a rule τ from P with $Pos_\tau = Pos_{\tau^*}$; and by adding, for every fact $\boldsymbol{g}$ in $V_n(Neg_\tau)$, a leaf under n with label $\boldsymbol{g}$ and sign '−'. Note that rules τ exist by definition of P^* and that $\mathcal{T}_e$ is an expansion tree because rules in P are safe and $\mathcal{T}_e^*$ is an expansion tree. Furthermore, $root_{\mathcal{T}_e} = root_{\mathcal{T}_e^*}$ and $Fringe^+_{\mathcal{T}_e} = Fringe^+_{\mathcal{T}_e^*}$.

It follows from the assumption that P is without explicit conflicts that $\mathcal{T}_e$ is free of fringe and inequality conflicts, thus with $root_{\mathcal{T}_e} \in q_P(Fringe^+_{\mathcal{T}_e})$, and from $q_P \in \mathcal{H}$ and homomorphism g with $g(Fringe^+_{\mathcal{T}_e}) = I$, that $\boldsymbol{f} = g(root_{\mathcal{T}_e^*}) = g(root_{\mathcal{T}_e}) \in q_P(I)$.

(2) Let I be an arbitrary instance over $in(P)$, and $\boldsymbol{f}$ an arbitrary fact in $q_{P^+}(I)$.

To show $\boldsymbol{f} \in q_P(I)$, we observe that the presence of $\boldsymbol{f}$ in $q_{P^+}(I)$ implies the existence of a proof tree $\mathcal{T}^+$ for P^+ with root $\boldsymbol{f}$ and $Fringe^+_{\mathcal{T}^+} \subseteq I$. By Proposition 3.6, there is an expansion tree $\mathcal{T}_e^+$ for P^+ that is without fringe and inequality conflicts; and a mapping g, with $g(\mathcal{T}_e^+) = \mathcal{T}$. We observe that $\mathcal{T}_e^+$ can be extended to an expansion tree $\mathcal{T}_e$ for P by adding '−' signed leafs. Indeed, for every intermediate vertex n in $\mathcal{T}_e^+$, say with label $\boldsymbol{f}_n$ and associated rule τ_n and valuation v_n, there is a rule $\tau \in P$ that differs from τ_n only w.r.t. the set of negated atoms (by construction of P^+). Due to safeness of rules, negated atoms do not introduce new variables that are not already in τ_n, hence, for every $A \in Neg_\tau$ we just add a new leaf under vertex n with label $v_n(A)$ and sign '−'.

Another consequence of the safeness of rules is that g is a total mapping for $\mathcal{T}_e$. Moreover, since P is free of fringe conflicts, and $g(\mathcal{T}_e)$ is a correctly defined candidate proof tree, it must be that $g(\mathcal{T}_e)$ is free of fringe conflicts. Since all inequalities in $\mathcal{T}_e$ already exist in $\mathcal{T}_e^+$, the fact that $g(\mathcal{T}_e^+)$ is free of inequality conflicts transfers to $g(\mathcal{T}_e)$.

As a consequence, $\mathcal{T} := g(\mathcal{T}_e)$ is a proof tree for $\boldsymbol{f}$ from P, with $Fringe^+_{\mathcal{T}} = Fringe^+_{\mathcal{T}^+}$, and $Fringe^+_{\mathcal{T}} \cap Fringe^-_{\mathcal{T}} = \emptyset$, thus $\boldsymbol{f} \in q_P(Fringe^+_{\mathcal{T}})$. We conclude from $q_P \in \mathcal{M}$ and $Fringe^+_{\mathcal{T}} = Fringe^+_{\mathcal{T}^+} \subseteq I$ that $\boldsymbol{f} \in q_P(I)$. ◀

Proposition 3.9(1) reveals a relation that was also observed by Feder and Vardi [12], albeit less explicit, to show the following result:

▶ **Theorem 3.10** ([12]). ***Sp-Datalog***$(\neq) \cap \mathcal{H} =$ ***Pos-Datalog***.

More precisely, the construction in [12] implies the next proposition, which together with Proposition 3.9 is a proof for Theorem 3.10.

▶ **Proposition 3.11.** *For every program $P \in$ Sp-Datalog$(\neq)$ there is a program $P' \in$ Sp-Datalog$(\neq)$ that is free of explicit conflicts and with $q_P = q_{P'}$.*

For an example of a program that is monotone but not conflict-free, one can take program $P_\triangle$ from Example 2.1.

▶ **Proposition 3.12.** *Program $P_\triangle$ expresses a monotone query and is not conflict-free.*

Since each program in *Pos-Datalog*$(\neq)$ is conflict-free by definition, it is immediate that a query $q \in$ **Sp-Datalog**$(\neq) \cap \mathcal{M}$ is in **Pos-Datalog**$(\neq)$ if and only if it can be expressed by a program in *Sp-Datalog*$(\neq)$ that is conflict-free. We conclude this section by showing that conflict freedom is a decidable property.

▶ **Theorem 3.13.** *The problem of deciding whether a given program in Sp-Datalog$(\neq)$ is conflict-free is polynomial-time equivalent with the non-satisfiability problem for programs in Pos-Datalog$(\neq)$:*

- *EXP-complete in general; and*
- *coNP-complete if a bound on the arity of relations is assumed.*

4 Semi-Positive Datalog

In this section, we answer one of the central questions of the paper and show that not all monotone queries expressible in semi-positive Datalog have an equivalent in positive Datalog.

▶ **Theorem 4.1.** ***Sp-Datalog***$(\neq) \cap \mathcal{M} \not\subseteq$ ***Pos-Datalog***$(\neq)$.

The proof for Theorem 4.1 is similar to a recent proof by Rudolph and Thomazo [23], which shows that the homomorphically closed queries expressible in order-invariant semi-positive Datalog do not all have an equivalent in order-invariant Datalog without negation (and without inequality). Before proceeding with the details, we first give a definition of order-invariant Datalog.

4.1 Order-Invariant Semi-Positive Datalog

Let σ be a database schema and $\sigma_\leq$ the extension of σ with relation names $\mathsf{Succ}^{(2)}$, $\mathsf{Min}^{(1)}$, and $\mathsf{Max}^{(1)}$. (We assume of course that Succ, Min and Max do not already occur in σ.) Then for an instance I over σ, by $I_\leq$ we denote the extension of I over $\sigma_\leq$ in which Succ is interpreted as the successor relation of some linear order over the active domain of I, and in which Min and Max are interpreted to contain exactly the minimal, respectively maximal, value that occurs in I according to the assumed linear order.

An *order-invariant Sp-Datalog$(\neq)$ program P* over schema σ then is defined as an *Sp-Datalog*$(\neq)$ program, say P', over schema $\sigma_\leq$, whose output is independent of the chosen linear order. That is, $q_{P'}(I_\leq) = q_{P'}(I'_\leq)$, for every instance I over σ and pair of extensions $I_\leq$ and $I'_\leq$ of I. Due to the latter, the semantics of q_P itself can be defined in terms of instances I over σ, as $q_P(I) := q_P(I_\leq)$ for arbitrary extension $I_\leq$ of I. Henceforth, we refer by *Sp-Datalog*$(\leq, \neq)$ to the class of order-invariant *Sp-Datalog*$(\neq)$ programs.

4.2 The perfect Matching Problem over Ordered Graphs

The query q_{PM} that we use to show **Sp-Datalog**$(\neq) \cap \mathcal{M} \subsetneq$ **Pos-Datalog**$(\neq)$ expresses a variant of the perfect matching problem. Given a graph $G = (V, E)$, the *perfect matching* problem asks whether a subset $M \subseteq E$ of edges exists such that every vertex in V is incident to exactly one edge in M. For the definition of q_{PM}, consider schemas $\sigma_1 := \{\mathsf{Edge}^{(2)}, \mathsf{Next}^{(2)}, \mathsf{First}^{(1)}, \mathsf{Last}^{(1)}\}$ and $\sigma_2 := \{\mathsf{Output}^{(0)}\}$. Given an instance I over σ_1, we denote by G_e the graph obtained by interpreting relation Edge as edge relation and its set of end-points as vertices. Graph G_n is defined analogously, by interpreting relation Next as edge relation and its set of end-points as vertices. We say that a vertex has label `first` if its associated value is in relation First and that it has label `last` if its value is in relation Last.

▶ **Definition 4.2** (Separating Query). *Let q_{PM} be the boolean query over input schema σ_1 and output schema σ_2, and with* Output() $\in q_{PM}(I)$ *if (and only if) one of the following conditions is true:*

1. *At least two different vertices have label* `first` *or* `last`*;*
2. *Some vertex in G_n has* either *label* `first` *and an incoming edge, label* `last` *and an outgoing edge, two incoming edges, two outgoing edges, or a self-loop; or*
3. *For set C, defined as the vertices in connected components of G_e that connect a vertex with label* `first` *to one with label* `last`*, graph G_e induced by the vertices in C has a perfect matching.*

Next, we show that q_{PM} has the desired properties.

▶ **Proposition 4.3.** *The following properties are true for q_{PM}:*

1. *q_{PM} is in **Sp-Datalog**$(\neq)$;*
2. *q_{PM} is monotone; and*
3. *q_{PM} is not in **Pos-Datalog**$(\neq)$.*

Proof Sketch. (1) Since the perfect matching problem is well-known to be in PTIME, we can assume an implementation in *Sp-Datalog*$(\neq)$. The latter is due to another well-known result, that the language *Sp-Datalog*$(\leq, \neq)$ captures exactly the PTIME computable queries [2].

To write a program in *Sp-Datalog*$(\neq)$ that expresses q_{PM}, we make use of program P (in which the relations Succ, Min, and Max are now intensional and no longer interpreted), and feed it a conservative fragment of the extensional relations Next, First, and Last.

(2) Monotonicity can be verified easily from Definition 4.2.

(3) The proof is analogous to a recent proof by Rudolph and Thomazo [23] for the statement **Sp-Datalog**$(\leq) \cap \mathcal{H} \subsetneq$ **Pos-Datalog**$(\leq)$. While our query is slightly different to the query used in [23] and admits inequalities, it uses the same key ingredients:

(a) A result by Razborov [21], which states that no family of monotone boolean circuits exists that answers the perfect matching problem and has circuits of polynomial size in the number of input gates.

(b) The existence of an algorithm that converts programs in *Pos-Datalog*$(\neq)$ that express q_{PM} into a family of monotone boolean circuits that answers the perfect matching problem and has circuits of polynomial size in the number of input gates. ◀

5 Semi-Positive Datalog without Inequalities

This section is devoted to showing Theorem 5.1.

▶ **Theorem 5.1.** ***Sp-Datalog*** $\cap \mathcal{M} =$ ***Pos-Datalog***.

Theorem 5.1 is a consequence of the observation that, for semi-positive Datalog without inequalities, the fragment of monotone queries and the fragment of queries preserved under homomorphisms collapses (cf. Proposition 5.2). That is, **Sp-Datalog** $\cap\, \mathcal{M} =$ **Sp-Datalog** $\cap\, \mathcal{H}$. Given this observation, Theorem 5.1 follows directly from Theorem 3.10.

▶ **Proposition 5.2.** *For a program $P \in$ Sp-Datalog, $q_P \in \mathcal{M}$ implies $q_P \in \mathcal{H}$.*

Proof. We show that for an arbitrary fact $\boldsymbol{f}$, instance I, and homomorphism h from I to $h(I)$, the fact $h(\boldsymbol{f})$ is in $q_P(h(I))$. The proof is by an iterative procedure that searches for an instance $I^{(i)}$, with the following properties:

1. There is a proof tree $\mathcal{T}$ of $\boldsymbol{f}$ from $I^{(i)}$ and P;
2. $adom(I^{(i)}) = adom(I)$;
3. $h(I^{(i)}) = h(I)$; and
4. $h(Fringe^{-}_{\mathcal{T}}) \cap h(Fringe^{+}_{\mathcal{T}}) = \emptyset$.

Suppose that the described instance $I^{(i)}$ exists, then the combination of Property (1) and Property (2) allows to apply homomorphism h to all valuations and facts associated to vertices in $\mathcal{T}$. The result is a candidate proof tree $\mathcal{T}'$ from P with $Fringe^{+}_{\mathcal{T}'} \subseteq h(I^{(i)})$ and $root_{\mathcal{T}'} = h(\boldsymbol{f})$. Since rules in P have no inequalities, $\mathcal{T}'$ is without inequality conflicts. Property (4) implies that $\mathcal{T}'$ is also without fringe conflicts, thus $\mathcal{T}'$ is a proof tree of $h(\boldsymbol{f})$ from P and $Fringe^{+}_{\mathcal{T}'}$ witnessing $h(\boldsymbol{f}) \in q_P(Fringe^{+}_{\mathcal{T}'})$. Now, the desired result $h(\boldsymbol{f}) \in q_P(h(I))$ follows from Property (3), implying $Fringe^{+}_{\mathcal{T}'} \subseteq h(I^{(i)}) = h(I)$, and $q_P \in \mathcal{M}$.

It remains to describe the procedure to find $I^{(i)}$, which uses an inductive argument taking conditions (1), (2), and (3) as invariants over the tentative instances that are being considered. As base case, we observe that the three invariants are true on I itself, by taking as $\mathcal{T}$ the proof tree of $\boldsymbol{f}$ from I and P. We now refer to I as $I^{(0)}$.

If Property (4) is true on the currently considered instance $I^{(i)}$, then we terminate the procedure. Otherwise, (†) there must be a fact $\boldsymbol{g} \in Fringe^{-}_{\mathcal{T}}$, such that $h(\boldsymbol{g}) \in h(Fringe^{+}_{\mathcal{T}}) \cap h(Fringe^{-}_{\mathcal{T}})$, with $\mathcal{T}$ the proof tree as defined by Property (1). It also follows from Property (1) that $\boldsymbol{g} \notin Fringe^{+}_{\mathcal{T}} \subseteq I^{(i)}$.

We now construct a new instance $I^{(i+1)}$ by adding $\boldsymbol{g}$ to $I^{(i)}$. Clearly, Property (3) is true, since $h(\boldsymbol{g}) \in h(Fringe^{+}_{\mathcal{T}}) \subseteq h(I^{(i)}) = h(I)$, which implies $h(I^{(i+1)}) = h(I^{(i)}) = h(I)$. Property (2) is straightforward as well, since the safeness of rules in P and $\boldsymbol{g} \in Fringe^{-}_{\mathcal{T}}$ imply $adom(\boldsymbol{g}) \subseteq adom(Fringe^{+}_{\mathcal{T}}) \subseteq adom(I^{(i)}) = adom(I)$. Finally, $I^{(i)} \subseteq I^{(i+1)}$ and $q_P \in \mathcal{M}$ imply $\boldsymbol{f} \in q_P(I^{(i+1)})$, which means that Property (1) is also as well.

Since the active domain of I is fixed and the number of facts $\boldsymbol{g} \notin I$ with $adom(\boldsymbol{g}) \subseteq adom(I)$ is finite, eventually $I^{(i)}$ cannot grow further and (†) must fail. From this, we conclude that the desired instance $I^{(i)}$ exists. ◀

6 Negation-Bounded Datalog

In Section 4, we have formally shown that some monotone queries in **Sp-Datalog**($\neq$) have no equivalent in *Pos-Datalog*($\neq$). This result implies that restricting ourselves to write programs in the positive variant of *Sp-Datalog*($\neq$) as a convenient way to write monotone programs in *Sp-Datalog*($\neq$), comes at the cost of some loss in expressive power. While a theoretician may be satisfied with this observation alone, a practitioner would likely wonder whether this gap matters in practice, for example, within a specific application domain. To help answer this question, an interesting direction is to consider conservative fragments of *Pos-Datalog*($\neq$) for which the monotone and positive fragment coincide. In Section 5, we have already seen that programs in *Sp-Datalog* have this property.

In this section, we define an orthogonal fragment, which we call negation-bounded Datalog.

▶ **Definition 6.1.** *Let $P \in$ Sp-Datalog($\neq$) be over some schema σ and $R \in \sigma$. Program P is* negation-bounded *if a positive integer k exists, such that for every instance I over σ and fact $\boldsymbol{f} \in q_P(I)$, there is a proof tree $\mathcal{T}$ of $\boldsymbol{f}$ from I and P with $|Fringe^{-}_{\mathcal{T}}| \leq k$.*

We immediately proceed with the main result of this section:

▶ **Theorem 6.2.** *For every program P in Sp-Datalog($\neq$) that is negation bounded, $q_P \in \mathcal{M}$ implies there is a program P' in Pos-Datalog($\neq$), with $q_{P'} = q_P$.*

For a proof of Theorem 6.2, we combine Proposition 3.9 with the below result.

▶ **Proposition 6.3.** *For every program P in Sp-Datalog($\neq$) that is negation bounded, there is a conflict-free program P' in Sp-Datalog($\neq$), with $q_P = q_{P'}$.*

The proof of Proposition 6.3 uses a technique that is inspired by the indexing technique in [12] to show Theorem 3.10, but rather than statically annotating relation names with associated dependencies, we encode indexes in a prefix of the intensional relations, which serve as pivot through the program evaluation. More precisely, program P' encodes the facts whose absence it observes while simulating program P in the prefix of intensional relation names and fires a rule of P in the simulation only if it is consistent with the index at hand. That is, if the index does not encode a fact that is required by the rule or any of its children. The latter is enforced via inequalities. (We note that similar techniques are used in, *e.g.*, [27, 8, 17].) As the proof of Proposition 6.3 is tedious, we illustrate the construction by an example.

▶ **Example 6.4.** Let P_Δ be again the program from Example 2.1. We notice that expansion trees for P_Δ have at most two negated atoms, corresponding, respectively, to the negated atom in the first and second rule of program P_Δ. Program P_Δ is thus clearly negation bounded. After applying the construction underlying Proposition 6.3, we obtain the following rules. First, two rules to collect in relation Adom the active domain of the input instance:

$$\mathsf{Adom(x) \leftarrow Edge(x,y).} \qquad\qquad \mathsf{Adom(x) \leftarrow Edge(y,x).}$$

Then, for every choice $\beta \in \{x \neq z, y \neq x\}$, $\gamma \in \{y \neq z, z \neq x\}$, $\chi \in \{x \neq y, y \neq z\}$ of inequalities, we consider variants of the T1 and T2 generating rules in P_Δ, in which their negated facts are encoded as a prefix in the head of the rule:

$$\mathsf{T1(z,x,x_2,y_2,x,y) \leftarrow Edge(x,y), Edge(y,z), \neg Edge(z,x), Adom(x_2), Adom(y_2)}, \beta, \gamma.$$
$$\mathsf{T2(x_1,y_1,y,z,x,y) \leftarrow Edge(x,y), \neg Edge(y,z), Edge(z,x), Adom(x_1), Adom(y_1)}, \chi, \gamma.$$

Rules without negated atoms forward the prefix of body atoms that are over intensional relation names, or (if no intensional relation name occurs in the body) generate facts with arbitrary prefix:

$$\begin{aligned}\mathsf{Output'(x_1,y_1,x_2,y_2,x,y)} &\leftarrow \mathsf{Edge(x,y), Edge(y,z), Edge(z,x),}\\ &\qquad \mathsf{Adom(x_1), Adom(y_1), Adom(x_2), Adom(y_2).}\\ \mathsf{Output'(x_1,y_1,x_2,y_2,x,y)} &\leftarrow \mathsf{T1(x_1,y_1,x_2,y_2,x,y), T2(x_1,y_1,x_2,y_2,x,y), x \neq y.}\\ \mathsf{Output(x,y)} &\leftarrow \mathsf{Output'(x_1,y_1,x_2,y_2,x,y).}\end{aligned}$$

Finally, we have to deal also with proof trees for $P_\triangle$ that have no leafs with sign '$-$'. To support this case, we augment the program with all rules from $P_\triangle$ that are without negation:

$$\mathsf{Output}(\mathsf{x},\mathsf{y}) \leftarrow \mathsf{Edge}(\mathsf{x},\mathsf{y}), \mathsf{Edge}(\mathsf{y},\mathsf{z}), \mathsf{Edge}(\mathsf{z},\mathsf{x}).$$

It is easy to verify that the program consisting of the above listed rules is conflict-free, since every fringe conflict now implies an inequality conflict. Equivalence follows from the observation that the constructed program simulates $P_\triangle$ with some additional bookkeeping.

We remark that the concept negation boundedness is related to the well-known concept boundedness for Datalog programs: A program P in *Sp-Datalog*$(\neq)$ is *bounded* if there is a positive integer k such that, for every instance I and fact $\boldsymbol{f} \in q_P(I)$, there is a proof tree of $\boldsymbol{f}$ from I and P with depth at most k. Since the latter implies existence of a bound on the size of $\mathit{Fringe}^+_\mathcal{T}$, and thus on the domain of $\mathit{Fringe}^+_\mathcal{T} \cup \mathit{Fringe}^-_\mathcal{T}$, it follows directly that boundedness implies negation boundedness. Not surprisingly, the decision problem that asks whether a given program $P \in$ *Sp-Datalog*$(\neq)$ is negation bounded is undecidable.

▶ **Proposition 6.5.** *No algorithm exists that decides for an arbitrary program $P \in$ Sp-Datalog$(\neq)$ whether it is negation bounded.*

Analogously to the classical result that the class of bounded programs is equally expressive as *UCQ*$(\neg,\neq)$ (the subset of *Sp-Datalog*$(\neq)$ programs in which the body of rules are constructed solely out of extensional relation names), we have the following syntactical characterisation:

▶ **Proposition 6.6.** *For every negation bounded program P in Sp-Datalog$(\neq)$ there is an equivalent program P' in Sp-Datalog$(\neq)$ that has a stratification P_1, P_2, with $P_1 \in$ Pos-Datalog$(\neq)$ and $P_2 \in$ UCQ$(\neg,\neq)$ (with no intensional relation name of P_2 occuring in P_1).*

Finally, we remark that the class of negation bounded programs can be extended a little, for example, by requiring a bound on the number of negated atoms only for relation symbols that occur positively in the program; or, by extension, for relation symbols that are not excluded from generating fringe conflicts due to some other syntactic reason. It is currently unclear whether a more fundamental generalization of Theorem 5.1 and Theorem 6.2 exists.

7 Stratified Datalog

A *stratified Datalog program* P is a set of rules as defined in Section 2.3, for which a stratification exists in a sequence of disjoint subprograms $P_1, \ldots, P_m$, with the following constraints: Every intensional relation name R in P occurs as a head in at most one subprogram P_i (we refer to P_i as the stratum in which R is defined).

- If an intensional relation name occurs positively in the body of a rule in subprogram P_i, then it is defined in a subprogram P_j, with $j \leq i$.
- If an intensional relation name occurs negated in the body of a rule in subprogram P_i, then it is defined in a subprogram P_j, with $j < i$.

We denote the class of stratified Datalog programs by *Str-Datalog*$(\neq)$. Since the subprograms P_i can be considered semi-positive, the semantics is defined as follows: $q_P(I) = q_{P_m} \circ q_{P_{m-1}} \circ \cdots \circ q_{P_1}(I)$. Here, we assume that for the subprograms P_i, with $i < m$ all relation names are output relation names, and for P_m only the distinguished output relations as defined by P. Similarly as before, we write **Str-Datalog**$(\neq)$ to denote the class of all queries expressible by programs in *Str-Datalog*$(\neq)$.

The following corollary is a straightforward consequence of Theorem 4.1.

▶ **Corollary 7.1.** *$\textbf{Str-Datalog}(\neq) \cap \mathcal{M} \not\subseteq \textbf{Pos-Datalog}(\neq)$.*

Despite Theorem 5.1, the gap between the monotone and positive fragment of stratified Datalog remains also without the interpreted inequality relation. The proof argument combines Theorem 4.1 with the observation that inequality is expressible through negation over intensional relation names.

▶ **Theorem 7.2.** *$\textbf{Str-Datalog} \cap \mathcal{M} \not\subseteq \textbf{Pos-Datalog}(\neq)$.*

8 A Best-Effort Approach to Negation Elimination

In this section, we consider the scenario in which an arbitrary *Str-Datalog*($\neq$) program is given and we are interested in finding an equivalent program with better computational properties (i.e., with less exposure to negation). Here, a program without negation is the ideal. Unfortunately, results like Proposition 3.9 do not help much to find such a program:

Firstly, the question if a given program in semi-positive Datalog is monotone is undecidable. Therefore, we cannot automatically infer whether a given program is in one of the desired subclasses.

▶ **Proposition 8.1.** *Testing whether a program in Sp-Datalog is monotone is undecidable.*

Secondly, these results only indicate whether an ideal equivalent rewriting (*i.e.*, a rewriting to a positive program) "certainly exists" or "may not exist". They do not help, especially in the latter case, to find such a program. Thirdly, even if no equivalent positive program exists, we may still be interested in finding an equivalent program with less exposure to negation.

The section proceeds as follows: In Section 8.1, we define a formal cost measure that allows to compare programs with negation. In Sections 8.2 and 8.3, we describe best-effort approaches towards improving the cost of a program as defined by this cost measure.

8.1 Cost Measure

We base our cost measure on observations from distributed Datalog evaluation (cf. Section 1): In an asynchronous distributed context (*e.g.*, [19, 16]), deciding the absence of a fact is significantly more difficult than deciding its presence, as it requires a round of consensus between the participating machines. One way to translate this observation into a formal cost measure is by hypothesising a correlation between the time it takes to derive an output fact for the first time and the minimal number of negated facts that its proof trees admit.

Notice that, in this hypothesis, positive programs are a conservative ideal (because proof trees of positive programs admit no negated facts), but programs that admit negated atoms are not necessarily considered worse (*i.e.*, if the rules with negated atoms are redundant).

For a formal definition, let P be an arbitrary program in *Sp-Datalog*($\neq$). We call a proof tree $\mathcal{T}$ from P *minimal* if no other proof tree $\mathcal{T}'$ from P exists that agrees with $\mathcal{T}$ on the label of its root, and has $\mathit{Fringe}^+_{\mathcal{T}'} \subseteq \mathit{Fringe}^+_{\mathcal{T}}$ and $\mathit{Fringe}^-_{\mathcal{T}'} \subseteq \mathit{Fringe}^-_{\mathcal{T}}$. Clearly, for every instance I and fact $\boldsymbol{f} \in q_P(I)$, we can always assume that a witnessing proof tree $\mathcal{T}$ exists that is minimal. Now, for two programs $P_1, P_2 \in$ *Sp-Datalog*($\neq$), we write $\mathit{cost}(P_1) \leq \mathit{cost}(P_2)$ if for every minimal proof tree $\mathcal{T}_1$ for P_1, with root an output fact, there is a proof tree $\mathcal{T}_2$ for P_2 that agrees with $\mathcal{T}_1$ on the label of its root, and with $\mathit{Fringe}^+_{\mathcal{T}_2} \subseteq \mathit{Fringe}^+_{\mathcal{T}_1}$ and $\mathit{Fringe}^-_{\mathcal{T}_2} \subseteq \mathit{Fringe}^-_{\mathcal{T}_1}$. We write $\mathit{cost}(P_1) < \mathit{cost}(P_2)$ if $\mathit{cost}(P_1) \leq \mathit{cost}(P_2)$ and for at least one such pair of proof trees $\mathcal{T}_1$ and $\mathcal{T}_2$, we have $\mathit{Fringe}^-_{\mathcal{T}_2} \subsetneq \mathit{Fringe}^-_{\mathcal{T}_1}$. We notice that our cost measure is defined over *Sp-Datalog*($\neq$) programs only, as we will use it to compare single-stratum fragments of *Str-Datalog*($\neq$) programs.

As can be expected, deciding properties about $cost(\cdot)$ quickly become undecidable.

▶ **Proposition 8.2.** *No algorithm exists that can decide for arbitrary equivalent programs* $P_1, P_2 \in$ *Sp-Datalog*$(\neq)$ *if* $cost(P_1) < cost(P_2)$*.*

8.2 Containment Testing

We start with an exploration of containment tests. Given a program $P \in$ *Sp-Datalog*$(\neq)$, we call a program $P' \in$ *Sp-Datalog*$(\neq)$ a *superior equivalent of* P if it is obtained by removing (some) negated atoms in rules from P. Clearly, P^+ is the extreme case, but now we are interested in programs P' that remain equivalent to the original program P. We note that $q_P \subseteq q_{P'}$ holds for every superior equivalent P' of P (the proof is a simple generalization of the proof argument for Proposition 3.2). Unfortunately, the other direction is undecidable.

▶ **Proposition 8.3.** *No procedure exists that decides* $q_{P'} \subseteq q_P$ *for arbitrary programs* $P \in$ *Sp-Datalog*$(\neq)$ *and superiorequivalent* P' *of* P*.*

To overcome this limitation, we test for $UCQ(\neg, \neq)$ containment instead (which is coNEXP-complete [13], respectively Π_2^p-complete [26, 20], if the arity of relations is bounded).

To formulate the next proposition, we need some additional notation: Given a program $P \in$ *Sp-Datalog*$(\neq)$, we denote by $\#(P)$ the program in $UCQ(\neg, \neq)$ obtained by adding a prime to all relation names occurring in the heads of rules (*i.e.*, $\mathsf{T}(\mathsf{x}, \mathsf{y}) \leftarrow \mathsf{E}(\mathsf{x}, \mathsf{z}), \mathsf{T}(\mathsf{z}, \mathsf{y})$ becomes $\mathsf{T}'(\mathsf{x}, \mathsf{y}) \leftarrow \mathsf{E}(\mathsf{x}, \mathsf{y}), \mathsf{T}(\mathsf{z}, \mathsf{y})$). Then, for $P_1, P_2 \in$ *Sp-Datalog*$(\neq)$ we test $q_{\#(P_1)} \subseteq q_{\#(P_2)}$ instead of $q_{P_1} \subseteq \#(P_2)$. Alternatively, one can consider uniform containment [24], which means that containment is tested for the queries described by P_1 and P_2, but taking as input schema the set of all extensional and intensional relation names of P_1 (resp, P_2) and as output schema the set of all intensional relation names of P_1 (resp, P_2). For the bounded arity case, a Π_2^p-completeness result is known due to Eiter and Fink [11]. The complexity for the non-bounded case appears to be open (*albeit* at least coNEXP-hard due to the earlier mentioned result for $UCQ(\neg, \neq)$ containment).

▶ **Proposition 8.4.** *For a program* $P \in$ *Sp-Datalog*$(\neq)$ *and superior equivalent* P' *of* P*,* $\#(P') \subseteq \#(P)$ *implies* $q_{P'} = q_P$ *and* $cost(P') \leq cost(P)$*.*

Let P be a stratified Datalog program. Then Proposition 8.4 admits a naive procedure, which we call NEG-ELIM, that applies to every stratum of P the following steps:

1. Test for every combination of negated atoms whether q_P is contained in $q_{P'}$, with P' the superior equivalent of P in which the selected atoms are removed.
2. Choose the superior equivalent of P that minimizes the total number of negated atoms, among those for which the test succeeds.

▶ **Theorem 8.5.** *Procedure* NEG-ELIM *runs with exponential space (respectively polynomial space, if a bound on the arity of considered relations is assumed).*

We remark that the special structure of $\#(P)$ and $\#(P')$ (*i.e.*, P' is a superior equivalent of P), does not admit a more efficient containment test.

▶ **Proposition 8.6.** *Testing for an arbitrary program* $P \in$ *Sp-Datalog*$(\neq)$ *and superior equivalent* P' *of* P *whether* $q_{\#(P')} \subseteq q_{\#(P)}$ *is* coNEXP*-hard (respectively* Π_2^p*-hard, if a bound on the arities of considered relations is assumed).*

8.3 Rule Expansions

One way to make the procedure from the previous section more powerful, is by doing containment tests for partially expanded program.

Let $P \in \mathit{Sp\text{-}Datalog}(\neq)$ and $\tau \in P$. By $\mathit{exp}(P)$ we denote the set of all 1-step expansions of rules in P. That is, the set of rules obtain from P by considering all possible replacements of intensional atoms in rules by the bodies of rules whose head matches the respective atom (after doing the necessary variable renaming).

▶ **Proposition 8.7.** *For every $P \in \mathit{Sp\text{-}Datalog}(\neq)$, $q_{\mathit{exp}(P)} = q_P$ and $\mathit{cost}(\mathit{exp}(P)) = \mathit{cost}(P)$.*

▶ **Example 8.8.** For an example illustrating the use of expansions in combination with algorithm NEG-ELIM, consider the two stratum program P, whose second stratum is program P_Δ from Example 2.1, and whose first stratum is a program P' over schema $\sigma' := \{\mathsf{Arc}^{(2)}, \mathsf{Edge}^{(2)}, \mathsf{Adom}^{(1)}\}$ with the next rules:

$$\mathsf{Adom}(\mathsf{x}) \leftarrow \mathsf{Arc}(\mathsf{x}, \mathsf{y}).$$
$$\mathsf{Adom}(\mathsf{x}) \leftarrow \mathsf{Arc}(\mathsf{y}, \mathsf{x}).$$
$$\mathsf{Edge}(\mathsf{x}, \mathsf{y}) \leftarrow \neg\mathsf{Arc}(\mathsf{x}, \mathsf{y}), \mathsf{Adom}(\mathsf{x}), \mathsf{Adom}(\mathsf{y}).$$

Now consider the expansion $\mathit{exp}(P_\Delta)$ of the second stratum:

$$\mathsf{Output}(\mathsf{x}, \mathsf{y}) \leftarrow \mathsf{Edge}(\mathsf{x}, \mathsf{y}), \mathsf{Edge}(\mathsf{y}, \mathsf{z}), \mathsf{Edge}(\mathsf{z}, \mathsf{x}).$$
$$\mathsf{Output}(\mathsf{x}, \mathsf{y}) \leftarrow \mathsf{Edge}(\mathsf{x}, \mathsf{y}), \mathsf{Edge}(\mathsf{y}, \mathsf{z}), \neg\mathsf{Edge}(\mathsf{z}, \mathsf{x}), \mathsf{Edge}(\mathsf{x}, \mathsf{y}), \neg\mathsf{Edge}(\mathsf{y}, \mathsf{w}), \mathsf{Edge}(\mathsf{w}, \mathsf{x}), \mathsf{x} \neq \mathsf{y}.$$

While directly applying algorithm NEG-ELIM over P_Δ does not improve the program, an application over $\mathit{exp}(P_\Delta)$ finds a negation-free equivalent:

$$\mathsf{Output}(\mathsf{x}, \mathsf{y}) \leftarrow \mathsf{Edge}(\mathsf{x}, \mathsf{y}), \mathsf{Edge}(\mathsf{y}, \mathsf{z}), \mathsf{Edge}(\mathsf{z}, \mathsf{x}).$$
$$\mathsf{Output}(\mathsf{x}, \mathsf{y}) \leftarrow \mathsf{Edge}(\mathsf{x}, \mathsf{y}), \mathsf{Edge}(\mathsf{y}, \mathsf{z}), \mathsf{Edge}(\mathsf{x}, \mathsf{y}), \mathsf{Edge}(\mathsf{w}, \mathsf{x}), \mathsf{x} \neq \mathsf{y}.$$

Hence, while P is not monotone, and therefore has no equivalent in $\mathit{Pos\text{-}Datalog}(\neq)$, we do obtain an equivalent single-stratum program $P_1 \cup P_2'$.

9 Conclusion

Motivated by applications in network programming, we studied fundamental questions about the relationship between the monotone and positive fragments of several variants of Datalog with negation (an overview is given by Figure 1). We also showed how the amount of negation that such programs admit can be decreased independently of whether they are monotone.

Related to monotonicity is the concept preservation under extensions ($\mathcal{E}$). While it is known that $\mathit{Sp\text{-}Datalog}(\neq) \subseteq \mathcal{E}$ [3], to the best of our knowledge, it is still an open question whether $\mathit{Str\text{-}Datalog}(\neq) \cap \mathcal{E} \overset{?}{=} \mathit{Sp\text{-}Datalog}(\neq)$. The latter question is of particular interest, because $\mathcal{E}$ is another notion that is associated with coordination in distributed systems [7].

The techniques that we discuss in Section 8, to remove negation from stratified Datalog programs, are by no means exhaustive. We also do not provide formal guarantees on the effectiveness of the approach. An interesting question therefore is whether other decidable techniques exist that can be of use for this purpose. Additionally, it would be interesting to perform an experimental study to see if these techniques can be combined into an effective procedure that is of use for real-live programs. The techniques that we present in Section 8 aim for a best-effort approach to automatically reduce the need for consensus in a program,

which in the context of Datalog, translates to the elimination of (stratified) negation. While it is obvious that such a procedure cannot beat optimization by hand, we see it useful in complex systems, that a program may be composed out of multiple programs and views. Then, optimization of individual subprograms does not necessarily imply optimization of the program as a whole, and a best-effort approach may be the only way to achieve improvement.

References

1 Serge Abiteboul, Meghyn Bienvenu, Alban Galland, and Émilien Antoine. A rule-based language for web data management. In *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2011)*, pages 293–304, 2011. doi:10.1145/1989284.1989320.

2 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995. URL: http://webdam.inria.fr/Alice/.

3 Foto N. Afrati, Stavros S. Cosmadakis, and Mihalis Yannakakis. On Datalog vs. Polynomial Time. *Journal of Computer and System Sciences*, 51(2):177–196, 1995. doi:10.1006/jcss.1995.1060.

4 Miklós Ajtai and Yuri Gurevich. Monotone versus positive. *Journal of the ACM*, 34(4):1004–1015, 1987. doi:10.1145/31846.31852.

5 Miklós Ajtai and Yuri Gurevich. Datalog vs First-Order Logic. *J. Comput. Syst. Sci.*, 49(3):562–588, 1994. doi:10.1016/S0022-0000(05)80071-6.

6 Tom J. Ameloot, Jan Van den Bussche, William R. Marczak, Peter Alvaro, and Joseph M. Hellerstein. Putting logic-based distributed systems on stable grounds. *Theory and Practice of Logic Programming*, 16(4):378–417, 2016. doi:10.1017/S1471068415000381.

7 Tom J. Ameloot, Bas Ketsman, Frank Neven, and Daniel Zinn. Weaker Forms of Monotonicity for Declarative Networking: A More Fine-Grained Answer to the CALM-Conjecture. *ACM Transactions on Database Systems*, 40(4):21:1–21:45, 2016. doi:10.1145/2809784.

8 Tom J. Ameloot, Bas Ketsman, Frank Neven, and Daniel Zinn. Datalog Queries Distributing over Components. *ACM Transactions on Computational Logic*, 18(1):5:1–5:35, 2017. doi:10.1145/3022743.

9 Surajit Chaudhuri and Moshe Y. Vardi. On the Equivalence of Recursive and Nonrecursive Datalog Programs. In *Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 1992)*, pages 55–66, 1992.

10 Anuj Dawar and Stephan Kreutzer. On Datalog vs. LFP. In *Automata, Languages and Programming, 35th International Colloquium, (ICALP 2008)*, pages 160–171, 2008. doi:10.1007/978-3-540-70583-3_14.

11 Thomas Eiter and Michael Fink. Uniform Equivalence of Logic Programs under the Stable Model Semantics. In *Logic Programming, 19th International Conference (ICLP 2003)*, pages 224–238, 2003. doi:10.1007/978-3-540-24599-5_16.

12 Tomás Feder and Moshe Y. Vardi. Homomorphism Closed vs. Existential Positive. In *18th IEEE Symposium on Logic in Computer Science (LICS 2003)*, pages 311–320, 2003. doi:10.1109/LICS.2003.1210071.

13 Gaetano Geck, Bas Ketsman, Frank Neven, and Thomas Schwentick. Parallel-Correctness and Containment for Conjunctive Queries with Union and Negation. In *19th International Conference on Database Theory (ICDT 2016)*, pages 9:1–9:17, 2016. doi:10.4230/LIPIcs.ICDT.2016.9.

14 Jim N. Gray. Notes on data base operating systems. *Lecture Notes in Computer Science*, 60:393–481, 1978.

15 Rachid Guerraoui and Jingjing Wang. How Fast can a Distributed Transaction Commit? In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, (PODS'17)*, pages 107–122, 2017. doi:10.1145/3034786.3034799.

16 Joseph M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Record*, 39(1):5–19, 2010. doi:10.1145/1860702.1860704.

17 Bas Ketsman, Aws Albarghouthi, and Paraschos Koutris. Distribution Policies for Datalog. In *21st International Conference on Database Theory (ICDT 2018)*, pages 17:1–17:22, 2018. doi:10.4230/LIPIcs.ICDT.2018.17.

18 Phokion G. Kolaitis and Moshe Y. Vardi. On the Expressive Power of Datalog: Tools and a Case Study. *J. Comput. Syst. Sci.*, 51(1):110–134, 1995. doi:10.1006/jcss.1995.1055.

19 Boon Thau Loo, Tyson Condie, Minos N. Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking: language, execution and optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2006)*, pages 97–108, 2006. doi:10.1145/1142473.1142485.

20 Marie-Laure Mugnier, Geneviève Simonet, and Michaël Thomazo. On the complexity of entailment in existential conjunctive first-order logic with atomic negation. *Information and Computation*, 215:8–31, 2012. doi:10.1016/j.ic.2012.03.001.

21 Aleksandr A. Razborov. Lower bounds on the monotone complexity of some Boolean functions. *Doklady Akademii Nauk SSSR*, 281:798–801, 1985.

22 Benjamin Rossman. Homomorphism preservation theorems. *Journal of the ACM*, 55(3):15:1–15:53, 2008. doi:10.1145/1379759.1379763.

23 Sebastian Rudolph and Michaël Thomazo. Expressivity of Datalog Variants - Completing the Picture. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2016)*, pages 1230–1236, 2016. URL: https://hal.inria.fr/hal-01302832.

24 Yehoshua Sagiv. Optimizing Datalog Programs. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 1987)*, pages 349–362, 1987. doi:10.1145/28659.28696.

25 Alexei P. Stolboushkin. Finitely Monotone Properties. In *Proceedings, 10th Annual IEEE Symposium on Logic in Computer Science*, pages 324–330, 1995. doi:10.1109/LICS.1995.523267.

26 Jeffrey D. Ullman. Information integration using logical views. *Theoretical Computer Science*, 239(2):189–210, 2000. doi:10.1016/S0304-3975(99)00219-4.

27 Ouri Wolfson and Abraham Silberschatz. Distributed Processing of Logic Programs. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data (SIGMOD 1988)*, pages 329–336, 1988. doi:10.1145/50202.50242.

28 Daniel Zinn, Todd J. Green, and Bertram Ludäscher. Win-Move is Coordination-Free (Sometimes). *CoRR*, abs/1312.2919, 2013. arXiv:1312.2919.