Stratified Negation in Limit Datalog Programs

Mark Kaminski, Bernardo Cuenca Grau, Egor V. Kostylev, Boris Motik and **Ian Horrocks**
Department of Computer Science, University of Oxford, UK
{mark.kaminski, bernardo.cuenca.grau, egor.kostylev, boris.motik, ian.horrocks}@cs.ox.ac.uk

Abstract

There has recently been an increasing interest in declarative data analysis, where analytic tasks are specified using a logical language, and their implementation and optimisation are delegated to a general-purpose query engine. Existing declarative languages for data analysis can be formalised as variants of logic programming equipped with arithmetic function symbols and/or aggregation, and are typically undecidable. In prior work, the language of *limit programs* was proposed, which is sufficiently powerful to capture many analysis tasks and has decidable entailment problem. Rules in this language, however, do not allow for negation. In this paper, we study an extension of limit programs with stratified negation-as-failure. We show that the additional expressive power makes reasoning computationally more demanding, and provide tight data complexity bounds. We also identify a fragment with tractable data complexity and sufficient expressivity to capture many relevant tasks.

1 Introduction

Data analysis tasks are becoming increasingly important in information systems. Although these tasks are currently implemented using code written in standard programming languages, in recent years there has been a significant shift towards declarative solutions where the definition of the task is clearly separated from its implementation [Alvaro *et al.*, 2010; Markl, 2014; Seo *et al.*, 2015; Wang *et al.*, 2015; Shkapsky *et al.*, 2016; Kaminski *et al.*, 2017].

Languages for declarative data analysis are typically rule-based, and they have already been implemented in reasoning engines such as BOOM [Alvaro *et al.*, 2010], DeALS [Shkapsky *et al.*, 2016], Myria [Wang *et al.*, 2015], SociaLite [Seo *et al.*, 2015], Overlog [Loo *et al.*, 2009], Dyna [Eisner and Filardo, 2011], and Yedalog [Chin *et al.*, 2015].

Formally, such declarative languages can be seen as variants of logic programming equipped with means for capturing quantitative aspects of the data, such as arithmetic function symbols and aggregates. It is, however, well-known since the '90s that the combination of recursion with numeric computations in rules easily leads to semantic difficulties [Mumick *et al.*, 1990; Kemp and Stuckey, 1991; Beeri *et al.*, 1991; Van Gelder, 1992; Consens and Mendelzon, 1993; Ganguly *et al.*, 1995; Ross and Sagiv, 1997; Mazuran *et al.*, 2013], and/or undecidability of reasoning [Dantsin *et al.*, 2001; Kaminski *et al.*, 2017]. In particular, undecidability carries over to the languages underpinning the aforementioned reasoning engines for data analysis.

Kaminski *et al.* [2017] have recently proposed the language of *limit Datalog programs*—a decidable variant of negation-free Datalog equipped with arithmetic functions over the integers that is expressive enough to capture many data analysis tasks. The key feature of limit programs is that all intensional predicates with a numeric argument are *limit predicates*, the extension of which represents minimal (min) or maximal (max) bounds of numeric values. For instance, if we encode a weighted directed graph as facts over a ternary $edge$ predicate and a unary $node$ predicate in the obvious way, then the following rules encode the all-pairs shortest path problem, where the ternary min limit predicate d is used to encode the distance from any node to any other node in the graph as the length of a shortest path between them.

$$node(x) \to d(x, x, 0) \tag{1}$$

$$d(x, y, m) \wedge edge(y, z, n) \to d(x, z, m + n) \tag{2}$$

The semantics of min predicates is defined such that a fact $d(u, v, k)$ is entailed from these rules and a dataset if and only if the distance from u to v is at most k; as a result, all facts $d(u, v, k')$ with $k' \geq k$ are also entailed. This is in contrast to standard first order predicates, where there is no semantic relationship between $d(u, v, k)$ and $d(u, v, k')$. The intended semantics of limit predicates can be axiomatised using rules over standard predicates; in particular, our example limit program is equivalent to a standard logic program consisting of rules (1), (2), and the following rule (3), where d is now treated as a regular first-order predicate:

$$d(x, y, k) \wedge (k \leq k') \to d(x, y, k'). \tag{3}$$

Kaminski *et al.* [2017] showed that, under certain restrictions on the use of multiplication, reasoning (i.e., fact entailment) over limit programs is decidable and coNP-complete in data complexity; then, they proposed a practical fragment with tractable data complexity.

Limit Datalog programs as defined in prior work are, however, positive and hence do not allow for negation-as-failure in the body of rules. Non-monotonic negation applied to limit atoms can be useful, not only to express a wider range of data analysis tasks, but also to declaratively obtain solutions to problems where the cost of such solutions is defined by a positive limit program. For instance, our example limit program consisting of rules (1) and (2) provides the length of a shortest path between any two nodes, but does not provide access to any of the paths themselves—an issue that we will be able to solve using non-monotonic negation.

In this paper, we study the language of limit programs with stratified negation-as-failure. Our language extends both positive limit Datalog as defined in prior work and plain (function-free) Datalog with stratified negation. We argue that our language provides useful additional expressivity, but at the expense of increased complexity of reasoning; for programs with restricted use of multiplication, complexity jumps from coNP-completeness in the case of positive programs, to Δ_2^{P}-completeness for programs with stratified negation. We also show that the tractable fragment of positive limit programs defined in [Kaminski *et al.*, 2017] can be seamlessly extended with stratified negation while preserving tractability of reasoning; furthermore, the extended fragment is sufficiently expressive to capture the relevant data analysis tasks.

The proofs of all our results are given in the appendix.

2 Preliminaries

In this section we recapitulate the syntax and semantics of Datalog programs with integer arithmetic and stratified negation (see e.g., [Dantsin *et al.*, 2001] for an excellent survey).

Syntax We assume a fixed vocabulary of countably infinite, mutually disjoint sets of *predicates* equipped with non-negative arities, *objects*, *object variables*, and *numeric variables*. Each position $1 \leq i \leq n$ of an n-ary predicate is of either *object* or *numeric sort*. An *object term* is an object or an object variable. A *numeric term* is an integer, a numeric variable, or of the form $s_1 + s_2$, $s_1 - s_2$, or $s_1 \times s_2$ where s_1 and s_2 are numeric terms and $+$, $-$, and $\times$ are the standard *arithmetic functions*. A *constant* is an object or an integer. A *standard atom* is of the form $B(t_1, \dots, t_n)$, with B an n-ary predicate and each t_i a term matching the sort of the i-th position of B. A *(standard) positive literal* is a standard atom, and a *(standard) negative literal* is of the form $\mathsf{not}\,\alpha$, for α a standard atom. A *comparison atom* is of the form $(s_1 < s_2)$ or $(s_1 \leq s_2)$, with $<$ and $\leq$ the usual *comparison predicates* over the integers, and s_1 and s_2 numeric terms. We write $(s_1 \doteq s_2)$ as an abbreviation for $(s_1 \leq s_2) \wedge (s_2 \leq s_1)$. A term, atom or literal is *ground* if it has no variables.

A *rule* r has the form $\bigwedge_i \mu_i \wedge \bigwedge_j \beta_j \to \alpha$, where the *body* $\bigwedge_i \mu_i \wedge \bigwedge_j \beta_j$ is a possibly empty conjunction of standard literals μ_i and comparison atoms β_j, and the *head* α is a standard atom. We assume without loss of generality that standard body literals are function-free; indeed, a conjunction with a functional term s can be equivalently rewritten by replacing s with a fresh variable x and adding $(x \doteq s)$ to the conjunction. A rule r is *safe* if each object variable in r occurs in a positive literal in the body of r. A *ground instance of* r is obtained from r by substituting each variable by a constant of the right sort.

A *fact* is a rule with empty body and a function-free standard atom in the head that has no variables in object positions and no repeated variables in numeric positions. Intuitively, a variable in a fact says that the fact holds for every integer in the position. As a convention, we will omit $\to$ and use symbol ∞ instead of variables when writing facts. A *dataset* $\mathcal{D}$ is a finite set of facts. Dataset $\mathcal{D}$ is *ordered* if *(i)* it contains facts $\mathit{first}(a_1)$, $\mathit{next}(a_1, a_2)$, $\dots$, $\mathit{next}(a_{n-1}, a_n)$, $\mathit{last}(a_n)$ for some repetition-free enumeration $a_1, \dots, a_n$ of all objects in $\mathcal{D}$; and *(ii)* it contains no other facts over predicates first, next, and last. A *program* is a finite set of safe rules; without loss of generality we assume that distinct rules do not share variables. A predicate B is *intensional* (*IDB*) in a program $\mathcal{P}$ if B occurs in $\mathcal{P}$ in the head of a rule that is not a fact; otherwise, B is *extensional* (*EDB*) in $\mathcal{P}$. Program $\mathcal{P}$ is *positive* if it has no negative literals, and it is *semi-positive* if negation occurs only in front of EDB atoms. A *stratification* of $\mathcal{P}$ is a function λ mapping each predicate to a positive integer such that, for each rule with the head over a predicate A and each standard body literal μ over B, we have $\lambda(B) \leq \lambda(A)$ if μ is positive, and $\lambda(B) < \lambda(A)$ if μ is negative. Program $\mathcal{P}$ is *stratified* if it admits a stratification. Given a stratification λ, we write $\mathcal{P}[i]$ for the *i-th stratum of $\mathcal{P}$ over λ*—that is, the set of all rules in $\mathcal{P}$ whose head predicates A satisfy $\lambda(A) = i$. Note that each stratum is a semi-positive program.

Semantics A *(Herbrand) interpretation* I is a possibly infinite set of ground facts (i.e., facts without ∞). Interpretation I *satisfies* a ground atom α, written $I \models \alpha$, if either *(i)* α is a standard atom such that evaluation of the arithmetic functions in α under the usual semantics over integers produces a fact in I; or *(ii)* α is a comparison atom that evaluates to true under the usual semantics. Interpretation I *satisfies* a ground negative literal $\mathsf{not}\,\alpha$, written $I \models \mathsf{not}\,\alpha$, if $I \not\models \alpha$. The notion of satisfaction is extended to conjunctions of ground literals, rules, and programs as in first-order logic, with all variables in rules implicitly universally quantified. If I satisfies a program $\mathcal{P}$, then I is a *model* of $\mathcal{P}$. For I a Herbrand interpretation and $\mathcal{R}$ a (possibly infinite) semi-positive set of rules, let $\mathbf{S}_{\mathcal{R}}(I)$ be the set of facts α such that $\varphi \to \alpha$ is a ground instance of a rule in $\mathcal{R}$ and $I \models \varphi$. Given a program $\mathcal{P}$ and a stratification λ of $\mathcal{P}$, for each $i, j \geq 0$ we define interpretation I_i^j by induction on i and j:

$$I_0^j = I_i^0 = \emptyset; \quad I_{i+1}^{j+1} = \mathbf{S}_{\mathcal{P}[i+1] \cup I_i^\infty}(I_{i+1}^j); \quad I_i^\infty = \bigcup_{j \geq 0} I_i^j.$$

The *materialisation* $\mathbf{M}(\mathcal{P})$ of $\mathcal{P}$ is the interpretation I_k^∞, for k the greatest number such that $\mathcal{P}[k] \neq \emptyset$. The materialisation of a program does not depend on the chosen stratification. A stratified program $\mathcal{P}$ *entails* a fact α, written $\mathcal{P} \models \alpha$, if $\alpha' \in \mathbf{M}(\mathcal{P})$ for every ground instance α' of α. For positive programs, this definition coincides with the usual first-order notion of entailment: for $\mathcal{P}$ positive and α a fact, $\mathcal{P} \models \alpha$ if and only if $I \models \alpha$ holds for all $I \models \mathcal{P}$.

Reasoning We study the computational properties of checking whether $\mathcal{P} \cup \mathcal{D} \models \alpha$, for $\mathcal{P}$ a program, $\mathcal{D}$ a dataset, and α a fact. We are interested in *data complexity*, which assumes that only $\mathcal{D}$ and α form the input while $\mathcal{P}$ is fixed. Unless otherwise stated, all numbers in the input are coded in binary, and the *size* $\|\mathcal{P}\|$ of $\mathcal{P}$ is the size of its representation. Checking $\mathcal{P} \cup \mathcal{D} \models \alpha$ is undecidable even if the only arithmetic function in $\mathcal{P}$ is $+$ [Dantsin *et al.*, 2001] and predicates have at most one numeric position [Kaminski *et al.*, 2017].

We use standard definitions of the basic complexity classes such as P, NP, CONP, and FP. Given a complexity class C, P^C is the class of decision problems solvable in polynomial time by deterministic Turing machines with an oracle for a problem in C; functional class FP^C is defined similarly. Finally, Δ_2^{P} is a synonym for P^{NP}.

3 Stratified Limit Programs

We introduce *stratified limit programs* as a language that can be seen as either a semantic or a syntactic restriction of Datalog with integer arithmetic and stratified negation. Our language is also an extension of that in [Kaminski *et al.*, 2017] with stratified negation.

Definition 1. *A* stratified limit program *is a pair* $(\mathcal{P}, \tau)$ *where*

- $\mathcal{P}$ *is a stratified program where each predicate either has no numeric position, in which case it is an* object predicate*, or only its last position is numeric, in which case it is a* numeric predicate*, and*
- τ *is a partial function from numeric predicates to* $\{\mathsf{min}, \mathsf{max}\}$ *that is total on the IDB predicates in* $\mathcal{P}$ *and on predicates occurring in non-ground facts.*

A numeric predicate A is a min *(or* max*)* limit predicate *if* $\tau(A) = \mathsf{min}$ *(or* $\tau(A) = \mathsf{max}$*, respectively). Numeric predicates that are not limit predicates are* ordinary*. An atom, fact or literal is* numeric, limit*, etc. if so is the used predicate.*

All notions defined on ordinary Datalog programs $\mathcal{P}$ (such as EDB and IDB predicates, stratification, etc.) transfer to limit programs $(\mathcal{P}, \tau)$ by applying them to $\mathcal{P}$. We often abuse notation and write $\mathcal{P}$ instead of $(\mathcal{P}, \tau)$ when τ is clear from the context or immaterial. Whenever we consider a union of two limit programs, we silently assume that they coincide on τ. Finally, we denote $\leq$ (or $\geq$) by $\preceq_A$ if A is a max (or, respectively, min) limit predicate.

Intuitively, a limit fact $B(\vec{a}, k)$ says that the value of B for a tuple of objects $\vec{a}$ is k *or more*, if B is max, or k *or less*, if B is min. For example, a min limit fact $d(u, v, k)$ in our all-pairs shortest path example says that node v is reachable from node u via a path with cost k or less. The intended semantics of limit predicates can be axiomatised using standard rules as given next.

Definition 2. *An interpretation I* satisfies *a limit program* $(\mathcal{P}, \tau)$ *if it satisfies the program* $\mathcal{P} \cup \mathsf{ax}(\mathcal{P})$*, where* $\mathsf{ax}(\mathcal{P})$ *contains the following rule for each limit predicate A in $\mathcal{P}$:*

$$A(\vec{x}, m) \wedge (n \preceq_A m) \rightarrow A(\vec{x}, n).$$

The materialisation $\mathbf{M}(\mathcal{P}, \tau)$ *of* $(\mathcal{P}, \tau)$ *is* $\mathbf{M}(\mathcal{P} \cup \mathsf{ax}(\mathcal{P}))$*; and* $(\mathcal{P}, \tau)$ entails α*, written* $(\mathcal{P}, \tau) \models \alpha$*, if* $\alpha \in \mathbf{M}(\mathcal{P}, \tau)$.

We next demonstrate the use of stratified negation on examples. One of the main uses of negation of a limit atom is to 'access' the limit value (e.g., the length of a shortest path) attained by the atom in the materialisation of previous strata, and then exploit such values in further computations. To facilitate such use of negation in examples, we introduce a new operator as syntactic sugar in the language.

Definition 3. *The* least upper bound expression $\lceil A(\vec{s}, n) \rceil$ *of a* max *(or* min*) limit atom* $A(\vec{s}, n)$ *is the conjunction* $A(\vec{s}, n) \wedge \mathsf{not}\, A(\vec{s}, m) \wedge (m \doteq n + t)$ *where* $t = 1$ *(or* $t = -1$*, respectively) and m is a fresh variable.*

Clearly, $I \models \lceil A(\vec{a}, k) \rceil$ for I an interpretation and $A(\vec{a}, k)$ a ground atom if k is the limit integer such that $I \models A(\vec{a}, k)$.

Example 4. An input of the single-pair shortest path problem can be encoded in the obvious way as a dataset $\mathcal{D}_{sp}$ using a ternary ordinary numeric predicate $edge$ to represent the graph's weighted edges, and unary facts $source(u)$ and $target(v)$ to identify the source and target nodes u and v, respectively. The stratified limit program $\mathcal{P}_{sp}$ given next computes, together with $\mathcal{D}_{sp}$ (where all edge weights are positive), a DAG over a binary object predicate $sp\text{-}edge$ such that every maximal path in the DAG is a shortest path from u to v.

$$source(x) \rightarrow ds(x, 0) \tag{4}$$

$$ds(x, m) \wedge edge(x, y, n) \rightarrow ds(y, m + n) \tag{5}$$

$$\begin{aligned} &\lceil ds(x, m_1) \rceil \wedge \lceil ds(y, m_2) \rceil \wedge \\ &\quad edge(x, y, n) \wedge target(y) \wedge \\ &\qquad (m_1 + n \doteq m_2) \rightarrow sp\text{-}edge(x, y) \end{aligned} \tag{6}$$

$$\begin{aligned} &\lceil ds(x, m_1) \rceil \wedge \lceil ds(y, m_2) \rceil \wedge \\ &\quad edge(x, y, n) \wedge sp\text{-}edge(y, z) \wedge \\ &\qquad (m_1 + n \doteq m_2) \rightarrow sp\text{-}edge(x, y) \end{aligned} \tag{7}$$

The first stratum consists of rules (4) and (5), and computes the length of a shortest path from u to all other nodes using the min predicate ds; in particular, $\mathcal{P}_{sp} \cup \mathcal{D}_{sp} \models \lceil ds(v, k) \rceil$ if and only if k is the length of a shortest path from u to v. Then, in a second stratum, the program computes the $sp\text{-}edge$ predicate such that $\mathcal{P}_{sp} \cup \mathcal{D}_{sp} \models sp\text{-}edge(a, b)$ if and only if the edge (a, b) is part of a shortest path from u to v. $\lhd$

Example 5. The closeness centrality of a node in a strongly connected weighted directed graph G is a measure of how central the node is in the graph [Sabidussi, 1966]; variants of this measure are useful, for instance, for the analysis of market potential. Most commonly, closeness centrality of a node u is defined as $1/\sum_{v \text{ node in } G} d(u, v)$, where $d(u, v)$ is the length of a shortest path from u to v; the sum in the denominator is often called the *farness centrality* of v. We next give a limit program computing a node of maximal closeness centrality in a given directed graph. We encode a graph as an ordered dataset $\mathcal{D}_{cc}$ using, as before, a unary object predicate $node$ and a ternary ordinary numeric predicate $edge$. Program $\mathcal{P}_{cc}$ consists of rules (8)–(16), where d, $fness'$ and $fness$ are min predicates, and $centre'$ and $centre$ are object predicates.

$$node(x) \rightarrow d(x, x, 0) \tag{8}$$

$$d(x, y, m) \wedge edge(y, z, n) \rightarrow d(x, z, m + n) \tag{9}$$

$$first(y) \wedge d(x, y, n) \rightarrow fness'(x, y, n) \tag{10}$$

$$\begin{aligned} &next(y, z) \wedge \\ &fness'(x, y, m) \wedge d(x, z, n) \rightarrow fness'(x, z, m + n) \end{aligned} \tag{11}$$

$$fness'(x, y, n) \wedge last(y) \rightarrow fness(x, n) \tag{12}$$

$$first(x) \rightarrow centre'(x, x) \tag{13}$$

$$\begin{aligned} &next(x, y) \wedge centre'(x, z) \wedge \\ &\lceil fness(z, n) \rceil \wedge \lceil fness(y, m) \rceil \wedge \\ &\qquad (m < n) \rightarrow centre'(y, y) \end{aligned} \tag{14}$$

$$
\begin{aligned}
&next(x, y) \wedge centre'(x, z) \wedge \\
&\lceil fness(z, n) \rceil \wedge \lceil fness(y, m) \rceil \wedge \\
&\qquad (n \leq m) \rightarrow centre'(y, z)
\end{aligned} \tag{15}
$$

$$
centre'(x, z) \wedge last(x) \rightarrow centre(z) \tag{16}
$$

The first stratum consists of rules (8)–(12). Rules (8) and (9) compute the distance (length of a shortest path) between any two nodes. Rules (10)–(12) then compute the farness centrality of each node based on the aforementioned distances; for this, the program exploits the order predicates to iterate over the nodes in the graph while recording the best value obtained so far in the iteration using an auxiliary predicate $fness'$. In the second stratum (rules (13)–(16)), the program uses negation to compute the node of minimum farness centrality (and hence of maximum closeness centrality), which is recorded using the $centre$ predicate; the order is again exploited to iterate over nodes, and an auxiliary predicate $centre'$ is used to record the current node of the iteration and the node with the best centrality encountered so far. $\triangleleft$

4 Stratified Limit-Linear Programs

By results in [Kaminski *et al.*, 2017], checking fact entailment is undecidable even for positive limit programs. Essentially, this follows from the fact that checking rule applicability over a set of facts requires solving arbitrary non-linear inequalities over integers—that is, solving the 10th Hilbert problem, which is undecidable. To regain decidability, they proposed a restriction on positive limit programs, called *limit-linearity*, which ensures that every program satisfying the restriction can be transformed using a grounding technique so that all numeric terms in the resulting program are linear. In particular, this implies that rule applicability can be determined by solving a system of linear inequalities, which is feasible in NP. As a result, fact entailment for positive limit-linear programs is CONP-complete in data complexity.

We next extend the notion of limit-linearity to programs with stratified negation, and define semi-grounding as a way to simplify a limit-linear program by replacing certain types of variables with constants. We then prove that fact entailment is Δ_2^{P}-complete in data complexity for such programs. All programs in our previous examples are limit-linear as per the definition given next.

Definition 6. *A numeric variable n is* guarded *in a rule r of a stratified limit program if*

- *either n occurs in a positive ordinary literal in r;*
- *or the body of r contains the literals*

$$
A(\vec{s}, n_1), \quad \mathsf{not}\, A(\vec{s}, n_2), \quad (n_2 \doteq n_1 + t),
$$

where A is a max (or min) predicate, $t = 1$ (or $t = -1$, respectively), and $n \in \{n_1, n_2\}$.

Rule r is limit-linear *if each numeric term in r is of the form $s_0 + \sum_{i=1}^{n} s_i \times m_i$, where each m_i is a distinct numeric variable not occurring in r in a (positive or negative) ordinary numeric literal, term s_0 uses only variables occurring in a positive ordinary literal in r, and terms s_i with $i \geq 1$ use only variables that are guarded in r and do not use $+$. A* limit-linear *program contains only limit-linear rules.*

A rule r is semi-ground *if all variables in r are numeric and occur only in limit and comparison atoms. The* semi-grounding *of a program $\mathcal{P}$ is obtained by replacing, in every rule r in $\mathcal{P}$, each object variable and each numeric variable occurring in an ordinary numeric atom in r with a constant in $\mathcal{P}$ in all possible ways.*

It is easily seen that the semi-grounding of a limit-linear program $\mathcal{P}$ entails the same facts as $\mathcal{P}$ for every dataset. Furthermore, as in prior work, Definition 6 ensures that the semi-grounding of a positive limit-linear program contains only linear numeric terms; finally, for programs with stratified negation, it ensures that negation can be eliminated while preserving limit-linearity when the program is materialised stratum-by-stratum, as we will discuss in detail later on.

Decidability of fact entailment for positive limit-linear programs is established by first semi-grounding the program and then reducing fact entailment over the resulting program to the validity problem of Presburger formulas [Kaminski *et al.*, 2017]—that is, first-order formulas interpreted over the integers and composed using only variables, constants 0 and 1, functions $+$ and $-$, and the comparisons.

The extension of such a reduction to stratified limit programs, however, is complicated by the fact that in the presence of negation-as-failure, entailment no longer coincides with classical first-order entailment. We thus adopt a different approach, where we show decidability and establish data complexity upper bounds according to the following steps.

Step 1. We extend the results in [Kaminski *et al.*, 2017] for positive programs by showing that, for every positive limit-linear program $\mathcal{P}$ and dataset $\mathcal{D}$, we can compute in $\mathrm{FP}^{\mathrm{NP}}$ a finite representation of its (possibly infinite) materialisation $\mathbf{M}(\mathcal{P} \cup \mathcal{D})$ (see Lemma 8 and Corollary 9). This representation is called the *pseudo-materialisation* of $\mathcal{P} \cup \mathcal{D}$.

Step 2. We further extend the results in Step 1 to semi-positive limit-linear programs, where negation occurs only in front of EDB predicates. For this, we show that fact entailment for such programs can be reduced in polynomial time in the size of the data to fact entailment over semi-ground positive limit-linear programs by exploiting the notion of a *reduct* (see Definition 10 and Lemma 11). Thus, we can assume existence of an $\mathrm{FP}^{\mathrm{NP}}$ oracle O for computing the pseudo-materialisation of a semi-positive limit-linear program.

Step 3. We provide an algorithm (see Algorithm 1) that decides entailment of a fact α by a stratified limit-linear program $\mathcal{P}$ using oracle O from Step 2. The algorithm maintains a pseudo-materialisation J, which is initially empty and is constructed bottom-up stratum by stratum. In each step i, the algorithm updates the pseudo-materialisation by applying O to the union of the pseudo-materialisation for stratum $i-1$ and the rules in the i-th stratum. The final J, from which entailment of α is obtained, is computed using a constant number of oracle calls in the size of the data, which yields a Δ_2^{P} data complexity upper bound (Proposition 13 and Theorem 15).

In what follows, we specify each of these steps. We start by formally defining the notion of a *pseudo-materialisation* $\mathbf{P}(\mathcal{P})$ of a stratified limit program $\mathcal{P}$, which compactly represents the materialisation $\mathbf{M}(\mathcal{P})$. Intuitively, $\mathbf{M}(\mathcal{P})$ can be infinite because it can contain, for any limit predicate B and tuple of objects $\vec{a}$ of suitable arity, an infinite number of facts of the form $B(\vec{a}, k)$. However, if the materialisation has facts of this form, then either there is a limit value ℓ such that $B(\vec{a}, k) \in \mathbf{M}(\mathcal{P})$ for each $k \preceq_B \ell$ and $B(\vec{a}, k') \notin \mathbf{M}(\mathcal{P})$ for each $k' \succ_B \ell$, or $B(\vec{a}, k) \in \mathbf{M}(\mathcal{P})$ for every integer k. As argued in prior work, it then suffices for the pseudo-materialisation to contain only a single fact $B(\vec{a}, \ell)$ in the former case, or $B(\vec{a}, \infty)$ in the latter case.

Definition 7. *A* pseudo-interpretation *J is a set of facts such that ∞ occurs only in facts over limit predicates and $k = k'$ holds for all facts $B(\vec{a}, k)$ and $B(\vec{a}, k')$ in J with limit B.*

The pseudo-materialisation *of a limit program $\mathcal{P}$, written $\mathbf{P}(\mathcal{P})$, is the (unique) pseudo-interpretation such that*

1. *an object or ordinary numeric fact is contained in $\mathbf{P}(\mathcal{P})$ if and only if it is contained in $\mathbf{M}(\mathcal{P})$; and*
2. *for each limit predicate B, object tuple $\vec{a}$, and integer ℓ,*
   - *$B(\vec{a}, \ell) \in \mathbf{P}(\mathcal{P})$ if and only if $B(\vec{a}, \ell) \in \mathbf{M}(\mathcal{P})$ and $B(\vec{a}, k) \notin \mathbf{M}(\mathcal{P})$ for all $k \succ_B \ell$, and*
   - *$B(\vec{a}, \infty) \in \mathbf{P}(\mathcal{P})$ if and only if $B(\vec{a}, k) \in \mathbf{M}(\mathcal{P})$ for all integers k.*

We now strengthen the results in [Kaminski *et al.*, 2017] by establishing a bound on the size of pseudo-materialisations of positive, limit-linear programs.

Lemma 8. *Let $\mathcal{P}$ be a semi-ground, positive, limit-linear program, and let $\mathcal{D}$ be a limit dataset. Then $|\mathbf{P}(\mathcal{P} \cup \mathcal{D})| \leq |\mathcal{P} \cup \mathcal{D}|$ and the magnitude of each integer in $\mathbf{P}(\mathcal{P} \cup \mathcal{D})$ is bounded polynomially in the largest magnitude of an integer in $\mathcal{P} \cup \mathcal{D}$, exponentially in $|\mathcal{P}|$, and double-exponentially in $\max_{r \in \mathcal{P}} \|r\|_u$, where $\|r\|_u$ stands for the size of the representation of r assuming that all numbers take unit space.*

By Lemma 8, the pseudo-materialisation of $\mathcal{P} \cup \mathcal{D}$ contains at most linearly many facts; furthermore, the size of each such fact is bounded polynomially once $\mathcal{P}$ is considered fixed. Hence, the pseudo-materialisation of $\mathcal{P}$ can be computed in $\mathrm{FP}^{\mathrm{NP}}$ in data complexity, even if $\mathcal{P}$ is not semi-ground.

Corollary 9. *Let $\mathcal{P}$ be a positive, limit-linear program. Then the function mapping each limit dataset $\mathcal{D}$ to $\mathbf{P}(\mathcal{P} \cup \mathcal{D})$ is computable in* $\mathrm{FP}^{\mathrm{NP}}$ *in $\|\mathcal{D}\|$.*

In our second step, we extend this result to semi-positive programs. For this, we start by defining the notion of a reduct of a semi-positive limit-linear program $\mathcal{P}$. The reduct is obtained by first computing a semi-ground instance $\mathcal{P}'$ of $\mathcal{P}$ and then eliminating all negative literals in $\mathcal{P}'$ while preserving fact entailment. Intuitively, negative literals can be eliminated because they involve only EDB predicates; as a result, their extension can be computed in polynomial time from the facts in $\mathcal{P}$ alone. To eliminate a ground negative literal μ, it suffices to check whether μ is entailed by the facts in $\mathcal{P}$ and simplify all rules containing μ accordingly; in turn, limit literals involving a numeric variable m can be rewritten as comparisons of m with a constant computed from the facts in $\mathcal{P}$.

Definition 10. *Let $\mathcal{P}$ be a semi-positive, limit-linear program and let $\mathcal{D}$ be the subset of all facts in $\mathcal{P}$. The* reduct *of $\mathcal{P}$ is obtained by first computing the semi-grounding $\mathcal{P}'$ of $\mathcal{P}$ and then applying the following transformations to each rule $r \in \mathcal{P}'$ and each negative body literal μ in r:*

1. *if $\mu = \mathsf{not}\, \alpha$, for α a ground atom, delete r if $\mathcal{D} \models \alpha$, and delete μ from r otherwise,*
2. *if $\mu = \mathsf{not}\, A(\vec{a}, m)$ is a non-ground limit literal, then*
   - *delete r if $\mathcal{D} \models A(\vec{a}, k)$ for each integer k;*
   - *delete μ from r if $\mathcal{D} \not\models A(\vec{a}, k)$ for each k; and*
   - *replace μ in r with $(k \prec_A m)$ otherwise, where $\mathcal{D} \models \lceil A(\vec{a}, k) \rceil$.*

ALGORITHM 1:

Parameter: oracle O computing $\mathbf{P}(\mathcal{P}')$ for $\mathcal{P}'$ a semi-positive, limit-linear program
Input: stratified, limit-linear program $\mathcal{P}$, fact α
Output: true if $\mathcal{P} \models \alpha$

```
compute a stratification λ of P
J := ∅
for i := 1 to max{k | P[k] ≠ ∅} do
 |  J := O(P[i] ∪ J)
end
return true if α is satisfied in J and false otherwise
```

Note that semi-ground programs disallow non-ground negative literals over ordinary numeric predicates, which is why these are not considered in Definition 10. As shown by the following lemma, reducts allow us to reduce fact entailment for semi-positive, limit-linear programs to semi-ground, positive, limit-linear programs.

Lemma 11. *For $\mathcal{P}$ a semi-positive, limit-linear program and $\mathcal{D}$ a limit dataset, $\mathcal{P}'$ the reduct of $\mathcal{P} \cup \mathcal{D}$, and α a fact, we have $\mathcal{P} \cup \mathcal{D} \models \alpha$ if and only if $\mathcal{P}' \models \alpha$. Moreover $\mathcal{P}'$ can be computed in polynomial time in $\|\mathcal{D}\|$, $\|\mathcal{P}'\|$ is polynomially bounded in $\|\mathcal{D}\|$, and $\max_{r \in \mathcal{P}'} \|r\|_u \leq \max_{r \in \mathcal{P} \cup \mathcal{D}} \|r\|_u$.*

The results in Lemma 8 and Lemma 11 imply that the pseudo-materialisation of a semi-positive, limit-linear program can be computed in $\mathrm{FP}^{\mathrm{NP}}$ in data complexity.

Lemma 12. *Let $\mathcal{P}$ be a semi-positive, limit-linear program. Then the function mapping each limit dataset $\mathcal{D}$ to $\mathbf{P}(\mathcal{P} \cup \mathcal{D})$ is computable in* $\mathrm{FP}^{\mathrm{NP}}$ *in $\|\mathcal{D}\|$.*

We are now ready to present Algorithm 1, which decides entailment of a fact α by a stratified limit-linear program $\mathcal{P}$. The algorithm uses an oracle O for computing the pseudo-materialisation of a semi-positive program. The existence of such oracle and its computational bounds are ensured by Lemma 12. Algorithm 1 constructs the pseudo-materialisation $\mathbf{P}(\mathcal{P})$ of $\mathcal{P}$ stratum by stratum in a bottom-up fashion. For each stratum i, the algorithm uses oracle O to compute the pseudo-materialisation of the program consisting of the rules in the current stratum and the facts in the pseudo-materialisation computed for the previous stratum. Once $\mathbf{P}(\mathcal{P})$ has been constructed, entailment of α is checked directly over $\mathbf{P}(\mathcal{P})$.

Correctness of the algorithm is immediate by the properties of O and the correspondence between pseudo-materialisations and materialisations. Moreover, if oracle O runs in FP^C in data complexity, for some complexity class C, then it can only return a pseudo-interpretation that is polynomially bounded in data complexity; as a result, Algorithm 1 runs in P^C since the number of strata of $\mathcal{P}$ does not depend on the input dataset.

Proposition 13. *If oracle O is computable in* FP^C *in data complexity, then Algorithm 1 runs in* P^C *in data complexity.*

The following upper bound immediately follows from the correctness of Algorithm 1 and Proposition 13.

Lemma 14. *For $\mathcal{P}$ a stratified, limit-linear program and α a fact, deciding $\mathcal{P} \models \alpha$ is in Δ_2^{P} in data complexity.*

The matching lower bound is obtained by reduction from the ODDMINSAT problem [Krentel, 1988]. An instance $\mathcal{M}$ of ODDMINSAT consists of a repetition-free tuple of variables $\langle x_N, \ldots, x_0 \rangle$ and a satisfiable propositional formula φ over these variables. The question is whether the truth assignment σ satisfying φ for which the tuple $\langle \sigma(x_N), \ldots, \sigma(x_0) \rangle$ is lexicographically minimal, assuming $false < true$, among all satisfying truth assignments of φ has $\sigma(x_0) = true$. In our reduction, $\mathcal{M}$ is encoded as a dataset $\mathcal{D}_\mathcal{M}$ using object predicates or and not to encode the structure of φ and numeric predicates to encode the order of variables in $\langle x_N, \ldots, x_0 \rangle$; a fixed, two-strata program $\mathcal{P}_{modd}$ then goes through all assignments σ in the ascending lexicographic order and evaluates the encoding of φ on σ until it finds some σ that makes φ true; $\mathcal{P}_{modd}$ then derives fact $minOdd$ if and only if $\sigma(x_0) = true$. Thus, $\mathcal{P}_{modd} \cup \mathcal{D}_\mathcal{M} \models minOdd$ if and only if $\mathcal{M}$ belongs to the language of ODDMINSAT.

Theorem 15. *For $\mathcal{P}$ a stratified, limit-linear program and α a fact, deciding $\mathcal{P} \models \alpha$ is Δ_2^{P}-complete in data complexity. The lower bound holds already for programs with two strata.*

5 A Tractable Fragment

Tractability in data complexity is an important requirement in data-intensive applications. In this section, we propose a syntactic restriction on stratified, limit-linear programs that is sufficient to ensure tractability of fact entailment in data complexity. Our restriction extends that of *type consistency* in prior work to account for negation. The programs in Examples 4 and 5 are type-consistent.

Definition 16. *A semi-ground, limit-linear rule r is* type-consistent *if*

- *each numeric term t in r is of the form $k_0 + \sum_{i=1}^{n} k_i \times m_i$ where k_0 is an integer and each k_i, $1 \leq i \leq n$, is a nonzero integer, called the* coefficient of variable m_i in t*;*
- *each numeric variable occurs in exactly one standard body literal;*
- *each numeric variable in a negative literal is guarded;*
- *if the head $A(\vec{a}, s)$ of r is a limit atom, then each unguarded variable occurring in s with a positive (or negative) coefficient also occurs in the body in a (unique) positive limit literal that is of the same (or different, respectively) type (i.e.,* min *vs.* max*) as A;*
- *for each comparison $(s_1 < s_2)$ or $(s_1 \leq s_2)$ in r, each unguarded variable occurring in s_1 with a positive (or negative) coefficient also occurs in a (unique) positive* min *(or* max*, respectively) body literal, and each unguarded variable occurring in s_2 with a positive (or negative) coefficient occurs in a (unique) positive* max *(or* min*, respectively) body literal.*

A semi-ground, stratified, limit-linear program is type-consistent *if all of its rules are type-consistent. A stratified limit-linear program $\mathcal{P}$ is* type-consistent *if the program obtained by first semi-grounding $\mathcal{P}$ and then simplifying all numeric terms as much as possible is type-consistent.*

Similarly to type-consistency for positive programs, Definition 16 ensures that divergence of limit facts to ∞ can be detected in polynomial time when constructing a pseudo-materialisation (see [Kaminski *et al.*, 2017] for details). Furthermore, the conditions in Definition 16 have been crafted such that the reduct of a semi-positive type-consistent program (and hence of any intermediate program considered while materialising a stratified program) can be trivially rewritten into a positive type-consistent program. For this, it is essential to require a guarded use of negation (see third condition in Definition 16).

Lemma 17. *For $\mathcal{P}$ a semi-positive, type-consistent program and $\mathcal{D}$ a limit dataset, the reduct of $\mathcal{P} \cup \mathcal{D}$ is polynomially rewritable to a positive, semi-ground, type-consistent program $\mathcal{P}'$ such that, for each fact α, $\mathcal{P} \cup \mathcal{D} \models \alpha$ if and only if $\mathcal{P}' \models \alpha$.*

Lemma 17 allows us to extend the polytime algorithm in [Kaminski *et al.*, 2017] for computing the pseudo-materialisation of a positive type-consistent program to semi-positive programs, thus obtaining a tractable implementation of oracle O restricted to type-consistent programs. This suffices since Algorithm 1, when given a type-consistent program as input, only applies O to type-consistent programs. Thus, by Proposition 13, we obtain a polynomial time upper bound on the data complexity of fact entailment for type-consistent programs with stratified negation. Since plain Datalog is already P-hard in data complexity, this upper bound is tight.

Theorem 18. *For $\mathcal{P}$ a stratified, type-consistent program and α a fact, deciding $\mathcal{P} \models \alpha$ is* P*-complete in data complexity.*

Finally, as we show next, our extended notion of type consistency can be efficiently recognised.

Proposition 19. *Checking whether a stratified, limit-linear program is type-consistent is in* LOGSPACE.

6 Conclusion and Future Work

Motivated by declarative data analysis applications, we have extended the language of limit programs with stratified negation-as-failure. We have shown that the additional expressive power provided by our extended language comes at a computational cost, but we have also identified sufficient syntactic conditions that ensure tractability of reasoning in data complexity. There are many avenues for future work. First, it would be interesting to formally study the *expressive power* of our language. Since type-consistent programs extend plain (function-free) Datalog with stratified negation, it is clear that they capture P on ordered datasets [Dantsin *et al.*, 2001], and we conjecture that the full language of stratified limit-linear programs captures Δ_2^{P}. From a more practical perspective, we believe that limit programs can naturally express many tasks that admit a dynamic programming solution (e.g., variants of the knapsack problem, and many others). Conceptually, a dynamic programming approach can be seen as a three-stage process: first, one constructs an acyclic 'graph of subproblems' that orders the subproblems from smallest to largest; then, one computes a shortest/longest path over this graph to obtain the value of optimal solutions; finally, one backwards-computes the actual solution by tracing back in the graph. Capturing the third stage seems to always require non-monotonic negation (as illustrated in our path computation example), whereas the first stage may or may not require it depending on the problem. Finally, the second stage can be realised with a (recursive) positive program. Second, our formalism should be extended with aggregate functions. Although certain forms of aggregation can be simulated using arithmetic functions and iterating over the object domain by exploiting the ordering, having aggregation explicitly would allow us to express certain tasks in a more natural way. Third, we would like to go beyond stratified negation and investigate the theoretical properties of limit Datalog under well-founded [Van Gelder *et al.*, 1991] or the stable model semantics [Gelfond and Lifschitz, 1988]. Finally, we plan to implement our reasoning algorithms and test them in practice.

Acknowledgments

This research was supported by the EPSRC projects DBOnto, MaSI^3, and ED^3.

References

[Alvaro *et al.*, 2010] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. BOOM analytics: exploring data-centric, declarative programming for the cloud. In *EuroSys 2010*, pages 223–236. ACM, 2010.

[Beeri *et al.*, 1991] Catriel Beeri, Shamim A. Naqvi, Oded Shmueli, and Shalom Tsur. Set constructors in a logic database language. *J. Log. Program.*, 10(3&4):181–232, 1991.

[Chin *et al.*, 2015] Brian Chin, Daniel von Dincklage, Vuk Ercegovac, Peter Hawkins, Mark S. Miller, Franz Josef Och, Christopher Olston, and Fernando Pereira. Yedalog: Exploring knowledge at scale. In *SNAPL 2015*, volume 32 of *LIPIcs*, pages 63–78. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.

[Chistikov and Haase, 2016] Dmitry Chistikov and Christoph Haase. The taming of the semi-linear set. In *ICALP*, volume 55 of *LIPIcs*, pages 128:1–128:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[Consens and Mendelzon, 1993] Mariano P. Consens and Alberto O. Mendelzon. Low complexity aggregation in GraphLog and Datalog. *Theor. Comput. Sci.*, 116(1):95–116, 1993.

[Dantsin *et al.*, 2001] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.

[Eisner and Filardo, 2011] Jason Eisner and Nathaniel Wesley Filardo. Dyna: Extending datalog for modern AI. In *Datalog 2010*, volume 6702 of *LNCS*, pages 181–220. Springer, 2011.

[Ganguly *et al.*, 1995] Sumit Ganguly, Sergio Greco, and Carlo Zaniolo. Extrema predicates in deductive databases. *J. Comput. Syst. Sci.*, 51(2):244–259, 1995.

[Gelfond and Lifschitz, 1988] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP 1988*, pages 1070–1080. MIT Press, 1988.

[Kaminski *et al.*, 2017] Mark Kaminski, Bernardo Cuenca Grau, Egor V. Kostylev, Boris Motik, and Ian Horrocks. Foundations of declarative data analysis using limit datalog programs. In *IJCAI 2017*, pages 1123–1130. ijcai.org, 2017.

[Kemp and Stuckey, 1991] David B. Kemp and Peter J. Stuckey. Semantics of logic programs with aggregates. In *ISLP 1991*, pages 387–401. MIT Press, 1991.

[Krentel, 1988] Mark W. Krentel. The complexity of optimization problems. *J. Comput. System Sci.*, 36(3):490–509, 1988.

[Loo *et al.*, 2009] 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. *Commun. ACM*, 52(11):87–95, 2009.

[Markl, 2014] Volker Markl. Breaking the chains: On declarative data analysis and data independence in the big data era. *PVLDB*, 7(13):1730–1733, 2014.

[Mazuran *et al.*, 2013] Mirjana Mazuran, Edoardo Serra, and Carlo Zaniolo. Extending the power of datalog recursion. *VLDB J.*, 22(4):471–493, 2013.

[Mumick *et al.*, 1990] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The magic of duplicates and aggregates. In *VLDB 1990*, pages 264–277. Morgan Kaufmann, 1990.

[Ross and Sagiv, 1997] Kenneth A. Ross and Yehoshua Sagiv. Monotonic aggregation in deductive databases. *J. Comput. System Sci.*, 54(1):79–97, 1997.

[Sabidussi, 1966] Gert Sabidussi. The centrality index of a graph. *Psychometrika*, 31(4):581–603, 1966.

[Seo *et al.*, 2015] Jiwon Seo, Stephen Guo, and Monica S. Lam. SociaLite: An efficient graph query language based on datalog. *IEEE Trans. Knowl. Data Eng.*, 27(7):1824–1837, 2015.

[Shkapsky *et al.*, 2016] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on Spark. In *SIGMOD 2016*, pages 1135–1149. ACM, 2016.

[Van Gelder *et al.*, 1991] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):620–650, 1991.

[Van Gelder, 1992] Allen Van Gelder. The well-founded semantics of aggregation. In *PODS 1992*, pages 127–138. ACM Press, 1992.

[Wang *et al.*, 2015] Jingjing Wang, Magdalena Balazinska, and Daniel Halperin. Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. *PVLDB*, 8(12):1542–1553, 2015.

A Proofs for Section 4

Before proceeding to the proofs of our theorems in the main body of the paper, we restate some notions from [Kaminski *et al.*, 2017]. All models of a limit program are easily seen to satisfy the following closure property.

Definition A.1. *An interpretation I is* limit-closed *(for a limit program $\mathcal{P}$) if, for each fact $B(\mathbf{a}, k) \in I$ where B is a limit predicate, $B(\mathbf{a}, k') \in I$ holds for each integer k' with $k' \preceq_B k$.*

There is a one-to-one correspondence between pseudo-interpretations and limit-closed interpretations, and thus each model of a program can be equivalently represented by a pseudo-interpretation.

Definition A.2. *A limit-closed interpretation I* corresponds *to a pseudo-interpretation J if the following conditions hold:*

- *an object or ordinary numeric fact is contained in J if and only if it is contained in I; and*
- *for each limit predicate B, each tuple of objects $\vec{b}$, and each integer ℓ, (i) $B(\vec{b}, k) \in I$ for all k if and only if $B(\vec{b}, \infty) \in J$, and (ii) $B(\vec{b}, \ell) \in I$ and $B(\vec{b}, k) \notin I$ for all $k \succ_B \ell$ and B is a limit predicate if and only if $B(\vec{b}, \ell) \in J$.*

Let J and J' be pseudo-interpretations corresponding to interpretations I and I'. Then, J satisfies *a ground atom α, written $J \models \alpha$, if $I \models \alpha$; J is a* pseudo-model *of a program $\mathcal{P}$, written $J \models \mathcal{P}$, if $I \models \mathcal{P}$; finally, $J \sqsubseteq J'$ holds if $I \subseteq I'$.*

Kaminski *et al.* [2017] then define an immediate consequence operator $\mathbf{T}_{\mathcal{P}}$ for positive limit programs that works on pseudo-interpretations and show that the pseudo-materialisation $\mathbf{P}(\mathcal{P})$ of a positive limit program $\mathcal{P}$ can be computed as the pseudo-interpretation J^∞ inductively defined as follows, where $\sup S$, for a set S of pseudo-interpretations, is the supremum of S w.r.t. $\sqsubseteq$:

$$J^0 = \emptyset \qquad J^{j+1} = \mathbf{T}_{\mathcal{P}}(J^j) \qquad J^\infty = \sup_{j \in \mathbb{N}} J^j$$

We call pseudo-interpretations J^i *partial pseudo-materialisations* of $\mathcal{P}$.

The CONP upper bound for fact entailment in [Kaminski *et al.*, 2017] is shown by a reduction to validity of Presburger formulas of a certain shape. We next extend this reduction $\mathsf{Pres}(\mathcal{P})$ as given in [Kaminski *et al.*, 2017] for a (semi-ground and positive) limit-linear program $\mathcal{P}$ to account for datasets involving ∞.

Definition A.3. *For each n-ary object predicate A, each $(n+1)$-ary ordinary numeric predicate B, each $(n+1)$-ary limit predicate C, each n-tuple of objects $\vec{a}$, and each integer k, let $\mathit{def}_{A\vec{a}}$, $\mathit{def}_{B\vec{a}k}$, $\mathit{def}_{C\vec{a}}$ and $\mathit{fin}_{C\vec{a}}$ be distinct propositional variables, and let $\mathit{val}_{C\vec{a}}$ a distinct integer variable.*

For $\mathcal{P}$ a semi-ground, positive, limit-linear program, $\mathsf{Pres}(\mathcal{P}) = \bigwedge_{r \in \mathcal{P}} \mathsf{Pres}(r)$ is the Presburger formula where $\mathsf{Pres}(r)$ is the formula (with the same quantifier block as r) that is obtained by replacing each atom α in r with its encoding $\mathsf{Pres}(\alpha)$ defined as follows:

- $\mathsf{Pres}(\alpha) = \alpha$ *if α is a comparison atom;*
- $\mathsf{Pres}(\alpha) = \mathit{def}_{A\vec{a}}$ *if α is an object atom of the form $A(\vec{a})$;*
- $\mathsf{Pres}(\alpha) = \mathit{def}_{B\vec{a}k}$ *if α is an ordinary numeric atom of the form $B(\vec{a}, s)$ where s is a ground numeric term evaluating to k;*[1]
- $\mathsf{Pres}(\alpha) = \mathit{def}_{C\vec{a}} \wedge (\neg \mathit{fin}_{C\vec{a}} \vee s \preceq_C \mathit{val}_{C\vec{a}})$ *if α is a limit atom of the form $C(\vec{a}, s)$ where $s \neq \infty$; and*
- $\mathsf{Pres}(\alpha) = \mathit{def}_{C\vec{a}} \wedge \neg \mathit{fin}_{C\vec{a}}$ *if α is a limit atom of the form $C(\vec{a}, \infty)$.*

Let J be a pseudo-interpretation, and let μ be an assignment of Boolean and integer variables. Then, J corresponds *to μ if all of the following conditions hold for all A, B, C, and $\vec{a}$ as specified above, for each integer $k \in \mathbb{Z}$:*

- *$\mu(\mathit{def}_{A\vec{a}}) = \mathit{true}$ if and only if $A(\vec{a}) \in J$;*
- *$\mu(\mathit{def}_{B\vec{a}k}) = \mathit{true}$ if and only if $B(\vec{a}, k) \in J$;*
- *$\mu(\mathit{def}_{C\vec{a}}) = \mathit{true}$ if and only if $C(\vec{a}, \infty) \in J$ or there exists $\ell \in \mathbb{Z}$ such that $C(\vec{a}, \ell) \in J$;*
- *$\mu(\mathit{fin}_{C\vec{a}}) = \mathit{true}$ and $\mu(\mathit{val}_{C\vec{a}}) = k$ if and only if $C(\vec{a}, k) \in J$.*

Note that k in Definition A.3 ranges over all integers (which excludes ∞), $\mu(\mathit{val}_{C\vec{a}})$ is equal to some integer k, and J is a pseudo-interpretation and thus cannot contain both $C(\vec{a}, \infty)$ and $C(\vec{a}, k)$; thus, $C(\vec{a}, \infty) \in J$ implies $\mu(\mathit{fin}_{C\vec{a}}) = \mathit{false}$.

The key property of the Presburger encoding in [Kaminski *et al.*, 2017] is established by the following lemma, which we easily re-prove for our variant of the encoding.

Lemma A.4. *Let J be a pseudo-interpretation and let μ be a variable assignment such that J corresponds to μ. Then,*

1. *$J \models \alpha$ if and only if $\mu \models \mathsf{Pres}(\alpha)$ for each ground atom α, and*
2. *$J \models r$ if and only if $\mu \models \mathsf{Pres}(r)$ for each semi-ground, positive rule r.*

[1] Note that all ordinary numeric atoms in $\mathcal{P}$ have this form since $\mathcal{P}$ is semi-ground.

Proof. Claim 1 follows analogously to the respective argument in [Kaminski *et al.*, 2017] except for having an extra case, namely $\alpha = C(\vec{a}, \infty)$, for C a limit predicate. The proof of this case is analogous but simpler to the case for $\alpha = C(\vec{a}, k)$ where $k \in \mathbb{Z}$. Claim 2 then follows from Claim 1 same as before. $\square$

Using Lemma A.4, Kaminski *et al.* [2017] establish the following correspondence between entailment for positive limit-linear programs and validity of Presburger sentences.

Lemma A.5. *For $\mathcal{P}$ a semi-ground, positive, limit-linear program and α a fact, there exists a Presburger sentence $\varphi = \forall \vec{x} \exists \vec{y}. \bigvee_{i=1}^{n} \psi_i$ that is valid if and only if $\mathcal{P} \models \alpha$. Each ψ_i is a conjunction of possibly negated atoms. Moreover, $|\vec{x}| + |\vec{y}|$ and each $\|\psi_i\|$ are bounded polynomially by $\|\mathcal{P}\| + \|\alpha\|$. Number n is bounded polynomially by $|\mathcal{P}|$ and exponentially by $\max_{r \in \mathcal{P}} \|r\|$. Finally, the magnitude of each integer in φ is bounded by the maximal magnitude of an integer in $\mathcal{P}$ and α.*

By a more precise analysis of the Presburger formulas in the proof of Lemma A.5, we can sharpen the bounds provided by the lemma as follows, where $\|r\|_u$ (resp. $\|\mathcal{P}\|$, $\|\varphi\|$, etc.) stands for the size of the representation of r (resp. $\mathcal{P}$, φ, etc.) assuming that all numbers take unit space.

Lemma A.6. *For $\mathcal{P}$ a semi-ground, positive, limit-linear program and α a fact, there exists a Presburger sentence $\varphi = \forall \vec{x} \exists \vec{y}. \bigvee_{i=1}^{n} \psi_i$ that is valid if and only if $\mathcal{P} \models \alpha$. Each ψ_i is a conjunction of possibly negated atoms. Moreover, $|\vec{x}| + |\vec{y}|$ is bounded polynomially in $\|\mathcal{P}\|_u$ and each $\|\psi_i\|_u$ is bounded polynomially in $\max_{r \in \mathcal{P}} \|r\|_u$. Number n is bounded polynomially in $|\mathcal{P}|$ and exponentially in $\max_{r \in \mathcal{P}} \|r\|_u$. Finally, the magnitude of each integer in φ is bounded by the maximal magnitude of an integer in $\mathcal{P}$ and α.*

Analogously to the notion of a model for an interpretation, we call With Lemma A.5 at hand, Kaminski *et al.* [2017] then show the following theorem, which bounds the magnitude of integers in counter-pseudo-models for entailment (the proof of the theorem adapts to our setting as is).

Theorem A.7. *For $\mathcal{P}$ a semi-ground, positive, limit-linear program, $\mathcal{D}$ a limit dataset, and α a fact, $\mathcal{P} \cup \mathcal{D} \not\models \alpha$ if and only if a pseudo-model J of $\mathcal{P} \cup \mathcal{D}$ exists where $J \not\models \alpha$, $|J| \leq |\mathcal{P} \cup \mathcal{D}|$, and the magnitude of each integer in J is bounded polynomially in the largest magnitude of an integer in $\mathcal{P} \cup \mathcal{D}$, exponentially in $|\mathcal{P}|$, and double-exponentially in $\max_{r \in \mathcal{P}} \|r\|$.*

Furthermore, the double-exponential bound in $\max_{r \in \mathcal{P}} \|r\|$ can be trivially sharpened to $\max_{r \in \mathcal{P}} \|r\|_u$ by employing Lemma A.6 in place of Lemma A.5. Building on the proof of Theorem A.7, we next prove the following stronger version, which bounds the size of pseudo-materialisations of semi-ground, positive, limit-linear programs.

Lemma 8. *Let $\mathcal{P}$ be a semi-ground, positive, limit-linear program, and let $\mathcal{D}$ be a limit dataset. Then $|\mathbf{P}(\mathcal{P} \cup \mathcal{D})| \leq |\mathcal{P} \cup \mathcal{D}|$ and the magnitude of each integer in $\mathbf{P}(\mathcal{P} \cup \mathcal{D})$ is bounded polynomially in the largest magnitude of an integer in $\mathcal{P} \cup \mathcal{D}$, exponentially in $|\mathcal{P}|$, and double-exponentially in $\max_{r \in \mathcal{P}} \|r\|_u$, where $\|r\|_u$ stands for the size of the representation of r assuming that all numbers take unit space.*

Proof. Let a be the maximal magnitude of an integer in $\mathcal{P} \cup \mathcal{D}$, $m = |\mathcal{P}|$, and $n = \max_{r \in \mathcal{P}} \|r\|_u$. Let $\mathcal{D}'$ be obtained from $\mathcal{D}$ by removing each fact that does not unify with an atom in $\mathcal{P}$ and let E be a fresh nullary predicate.

Clearly, we have $\mathbf{P}(\mathcal{P} \cup \mathcal{D}) = \mathbf{P}(\mathcal{P} \cup \mathcal{D}') \cup J_0$ where J_0 is the least pseudo-interpretation w.r.t. $\sqsubseteq$ such that $\{\alpha\} \sqsubseteq J_0$ for each $\alpha \in \mathcal{D} \setminus \mathcal{D}'$. Let φ be obtained from $\mathcal{P} \cup \mathcal{D}'$ and fact E analogously to the construction in the proof of Lemma A.6, but where each disjunct $(\neg fin_{C\vec{a}} \vee s \preceq_C val_{C\vec{a}})$ in $\mathsf{Pres}(\mathcal{P})$ is replaced by $\neg fin_{C\vec{a}}$ if $C(\vec{a}, \infty) \in \mathbf{P}(\mathcal{P} \cup \mathcal{D}')$ and by $s \preceq_C val_{C\vec{a}}$ if $C(\vec{a}, k) \in \mathbf{P}(\mathcal{P} \cup \mathcal{D}')$ for some $k \in \mathbb{Z}$. It is easy to see that every assignment corresponding to $\mathbf{P}(\mathcal{P} \cup \mathcal{D}')$ is a countermodel of φ. Therefore, since φ satisfies the same structural constraints as the formula in Lemma A.6, by an argument analogous to the one in the proof of Theorem A.7 we obtain that $\mathcal{P} \cup \mathcal{D}$ has a pseudo-model J such that $|J| \leq |\mathcal{P} \cup \mathcal{D}|$, the magnitude of each integer in J is bounded by some number ℓ that is polynomial in a, exponential in m, and double-exponential in n, and where, it holds that $C(\vec{a}, \infty) \in J$ if and only if $C(\vec{a}, \infty) \in \mathbf{P}(\mathcal{P} \cup \mathcal{D})$ for each limit predicate C and objects $\vec{a}$. Consequently, we have established that $\mathbf{P}(\mathcal{P} \cup \mathcal{D})$ has a pseudo-model J that satisfies the required bounds in the lemma. In what follows we use the fact that $\mathbf{P}(\mathcal{P} \cup \mathcal{D}) \sqsubseteq J$ to show that $\mathbf{P}(\mathcal{P} \cup \mathcal{D})$ also satisfies the bounds in the lemma.

Let us denote with J^j the partial pseudo-materialisation of $\mathcal{P} \cup \mathcal{D}$ for any $j \geq 0$ and hence, $\mathbf{P}(\mathcal{P} \cup \mathcal{D}) = J^\infty$. We start with the observation that ($\star$) the value of a number k in a limit fact $A(\vec{a}, k)$ can only increase with respect to $\preceq_A$ during the construction of $\mathbf{P}(\mathcal{P} \cup \mathcal{D})$. For instance, if $A(\vec{a}, k) \in J^j$, with A a max predicate, and $A(\vec{a}, k') \in J^{j+1}$, then $k' \geq k$. Let, $\ell_0 = \ell$ and, for $j > 0$, ℓ_j be the maximum between

- ℓ_{j-1},
- the maximal magnitude of a negative integer occurring in a max fact in J^j, and
- the maximal magnitude of a positive integer occurring in a min fact in J^j.

Numbers ℓ_j allow us to bound the integers produced by the immediate consequence operator $\mathbf{T}_{\mathcal{P}}$ applied to pseudo-interpretation J^j. Specifically, we argue that (♠) for each j and rule r with head $A(\mathbf{a}, s)$ for some s, we have

- $|\mathsf{opt}(r, J^j)| \leq n2^{O(n \log n)} \ell^{n+1} \ell_j$ if $A(\vec{a}, \infty) \notin \mathbf{P}(\mathcal{P} \cup \mathcal{D})$,
- $\mathsf{opt}(r, J^j) \geq -n2^{O(n \log n)} \ell^{n+1} \ell_j$ if A is a max predicate, and
- $\mathsf{opt}(r, J^j) \leq n2^{O(n \log n)} \ell^{n+1} \ell_j$ if A is a min predicate.

To see why this holds, consider a pseudo-interpretation J' obtained from J^j by replacing each $\max$ IDB fact $B(\vec{b}, k)$ with $B(\vec{b}, -\ell_j)$, and each $\min$ IDB fact $C(\vec{c}, k')$ with $C(\vec{c}, \ell_j)$. By construction, we have $\{\mathsf{opt}(r, J')\} \sqsubseteq \{\mathsf{opt}(r, J^j)\} \sqsubseteq J$ and hence $\mathsf{opt}(r, J') \preceq_A \mathsf{opt}(r, J^j) \preceq_A \mathsf{opt}(r, J)$ whenever $\mathsf{opt}(r, J')$ is defined. But since the magnitude of all numbers in J' is bounded by ℓ_j, by Proposition 3 in [Chistikov and Haase, 2016], $\mathcal{C}(r, J')$ has a solution where the maximal magnitude of all numbers is bounded by $2^{O(n \log n)}\ell^n \ell_j$, and hence the magnitude of the value k of s for this solution is bounded by $n2^{O(n \log n)}\ell^{n+1}\ell_j$ (unless the value of s is unbounded in $\mathcal{C}(r, J')$, in which case $\mathsf{opt}(r, J') = \mathsf{opt}(r, J)$ and we are done). The last two subclaims are immediate since $k \preceq_A \mathsf{opt}(r, J') \preceq_A \mathsf{opt}(r, J^j)$, and and the first claim follows since, additionally, $\mathsf{opt}(r, J^j) \preceq_A \mathsf{opt}(r, J)$, and $A(\vec{a}, \infty) \notin \mathbf{P}(\mathcal{P} \cup \mathcal{D})$ implies $|\mathsf{opt}(r, J)| \leq \ell = \ell_0$ by our assumptions about J.

From ($\star$) we can conclude that, for each j and k such that $j \leq k$ and $|J^j| = |J^k|$, we have $\ell_j = \ell_k$. Thus, whenever ℓ_j increases during the construction of $\mathbf{P}(\mathcal{P} \cup \mathcal{D})$, this must be because a rule has generated a fact $B(\vec{b}, i)$ where there was previously no fact over B and $\vec{b}$ in the partial pseudo-materialisation. The number of times this can happen is obviously bounded by m (i.e., the number of rules in $\mathcal{P}$). Furthermore, by (♠), whenever $\ell_j < \ell_{j+1}$, we have $\ell_{j+1} \leq n2^{O(n \log n)}\ell^{n+1}\ell_j$. Consequently, for every j, we have that $\ell_j \leq n^m 2^{O(mn \log n)}\ell^{m(n+1)+1}$

By (♠), we conclude that the maximal magnitude L of every integer in $\mathbf{P}(\mathcal{P} \cup \mathcal{D}) = J^\infty$ is bounded by $n^{m+1}2^{O((m+1)n \log n)}\ell^{(m+1)(n+1)+1}$. Clearly, L is polynomially bounded in a, exponentially in m, and double-exponentially in n since so is ℓ. $\square$

Lemma 11. *For $\mathcal{P}$ a semi-positive, limit-linear program and $\mathcal{D}$ a limit dataset, $\mathcal{P}'$ the reduct of $\mathcal{P} \cup \mathcal{D}$, and α a fact, we have $\mathcal{P} \cup \mathcal{D} \models \alpha$ if and only if $\mathcal{P}' \models \alpha$. Moreover $\mathcal{P}'$ can be computed in polynomial time in $\|\mathcal{D}\|$, $\|\mathcal{P}'\|$ is polynomially bounded in $\|\mathcal{D}\|$, and* $\max_{r \in \mathcal{P}'} \|r\|_u \leq \max_{r \in \mathcal{P} \cup \mathcal{D}} \|r\|_u$.

Proof. To show $\mathcal{P} \cup \mathcal{D} \models \alpha$ iff $\mathcal{P}' \models \alpha$, it suffices to argue that $\mathcal{P}' \models \alpha$ holds iff $\mathcal{P}'' \models \alpha$, for $\mathcal{P}''$ the semi-grounding of $\mathcal{P} \cup \mathcal{D}$.

Since $\mathcal{P}''$ is semi-positive and $\mathcal{P}'$ positive, w.l.o.g., we have $\mathbf{M}(\mathcal{P}') = I_1^\infty$, $\mathbf{M}(\mathcal{P}'') = H_2^\infty$. We show that, for each $i \in \mathbb{N}$, *(i)* $I_1^i \subseteq \mathbf{M}(\mathcal{P}'')$, and *(ii)* $H_2^i \subseteq \mathbf{M}(\mathcal{P}')$, by simultaneous induction on i, which implies the claim by the definition of entailment. Note that, for r a rule, we will denote the body of r as $\mathsf{b}(r)$.

For $i = 0$, the claim is trivial since $I_1^0 = H_2^0 = \emptyset$.

For $i > 0$, suppose first $\beta \in I_1^i$ for some β. We show $\beta \in \mathbf{M}(\mathcal{P}'')$. Since $\beta \in \mathbf{S}_{\mathcal{P}'}(I_1^{i-1})$, there is a rule $r' \in \mathcal{P}'$ such that, for some grounding σ, $I_1^{i-1} \models \mathsf{b}(r'\sigma)$. Moreover, by the inductive hypothesis, $I_1^{i-1} \subseteq \mathbf{M}(\mathcal{P}'')$. Let r'' be the rule in $\mathcal{P}''$ such that r' is obtained from r''. It suffices to show $\mathbf{M}(\mathcal{P}'') \models \mathsf{b}(r''\sigma)$. By construction, all literals in $\mathsf{b}(r')$ are positive and the only literals in $\mathsf{b}(r'') \setminus \mathsf{b}(r')$ are negative literals of the form $\mathsf{not}\,\alpha$, so, since $I_1^{i-1} \subseteq \mathbf{M}(\mathcal{P}'')$, it suffices to show that $\mathbf{M}(\mathcal{P}'') \models \mathsf{not}\,\alpha\sigma$ for each $\mathsf{not}\,\alpha \in \mathsf{b}(r'')$. We distinguish two cases.

If α is ground, we have $\alpha\sigma = \alpha$ and, by construction of r', we have $\mathcal{D}' \not\models \alpha$, where $\mathcal{D}'$ is the set of facts in $\mathcal{P} \cup \mathcal{D}$. Consequently, $\mathcal{P}'' \not\models \alpha$ since $\mathcal{P}''$ and $\mathcal{D}'$ coincide on facts and α must be EDB in $\mathcal{P}''$ (which is the case since $\mathcal{P}$ is semi-positive), and so $\mathbf{M}(\mathcal{P}'') \not\models \alpha$, and so $\mathbf{M}(\mathcal{P}'') \models \mathsf{not}\,\alpha = \mathsf{not}\,\alpha\sigma$, as required.

If α is non-ground, it must be a limit atom of the form $A(\vec{a}, m)$ (since $\mathcal{P}''$ is semi-ground and thus negative ordinary numeric literals contain no numeric variables). By construction of r', one of the following two cases must hold.

- $\mathcal{D}' \not\models A(\vec{a}, k)$ for each $k \in \mathbb{Z}$, and hence $\mathcal{D}' \not\models A(\vec{a}, m\sigma)$.
- $\mathcal{D}' \models \lceil A(\vec{a}, k) \rceil$ for some $k \in \mathbb{Z}$ and $(k \prec_A m) \in \mathsf{b}(r')$. Since $I_1^{i-1} \models \mathsf{b}(r'\sigma)$, we then have $k \prec_A m\sigma$, and hence $\mathcal{D}' \not\models A(\vec{a}, m\sigma)$.

Since A must be EDB in $\mathcal{P}''$, we then conclude $\mathbf{M}(\mathcal{P}'') \models \mathsf{not}\,\alpha\sigma$ analogously to before.

Next, suppose $\beta \in H_2^i$ for some β. We show $\beta \in \mathbf{M}(\mathcal{P}')$. Since $\beta \in \mathbf{S}_{\mathcal{P}''[2] \cup H_1^\infty}(H_2^{i-1})$, there is a rule $r'' \in \mathcal{P}''[2] \cup H_1^\infty$ such that, for some grounding σ, $H_2^{i-1} \models \mathsf{b}(r''\sigma)$. Moreover, by the inductive hypothesis, $H_2^{i-1} \subseteq \mathbf{M}(\mathcal{P}')$.

We distinguish two cases. If $r'' = \beta \in H_1^\infty$, it is easily seen that $\beta \in \mathbf{M}(\mathcal{P}')$ since, by construction, we have $\mathcal{P}''[1] \subseteq \mathcal{P}'$, and hence $H_1^\infty = \mathbf{M}(\mathcal{P}''[1]) \subseteq \mathbf{M}(\mathcal{P}')$ since $\mathcal{P}'$ is positive. Thus, w.l.o.g., suppose $r'' \in \mathcal{P}''[2]$. It then suffices to show that there is a rule $r' \in \mathcal{P}'$ obtained from r'' such that $\mathbf{M}(\mathcal{P}') \models \mathsf{b}(r'\sigma)$. Since $H_2^{i-1} \models \mathsf{b}(r''\sigma)$, we have $H_1^\infty \not\models \alpha\sigma$ for each negative literal $\mathsf{not}\,\alpha \in \mathsf{b}(r'')$, and hence also $\mathcal{D}' \not\models \alpha\sigma$. Consequently, rule r'' is not deleted by the transformation rules in Definition 10 but rather transformed to a positive rule r' such that the only literals in $\mathsf{b}(r') \setminus \mathsf{b}(r'')$ have the form $k \prec_A m$ such that $\mathsf{not}\,A(\vec{a}, m)$ is a non-ground limit literal in $\mathsf{b}(r'')$ and $\mathcal{D}' \models \lceil A(\vec{a}, k) \rceil$. Thus, since r' is positive and $H_2^{i-1} \subseteq \mathbf{M}(\mathcal{P}')$, it suffices to show $\mathbf{M}(\mathcal{P}') \models k \prec_A m\sigma$ for each such literal $(k \prec_A m) \in \mathsf{b}(r')$. This follows since, by construction and since $\mathcal{P}''$ is semi-positive, A is EDB in $\mathcal{P}'$, and hence $\mathbf{M}(\mathcal{P}) \models A(\vec{a}, s)$ holds for a term s if and only if $\mathcal{D}' \models A(\vec{a}, s)$; for each literal $(k \prec_A m) \in \mathsf{b}(r')$, we then have $\mathbf{M}(\mathcal{P}) \not\models A(\vec{a}, m\sigma)$ and $\mathbf{M}(\mathcal{P}) \models A(\vec{a}, k)$, which implies $\mathbf{M}(\mathcal{P}) \models k \prec_A m\sigma$, as required.

For the second claim, note that, by construction, $\|\mathcal{P}'\|$ is bounded from above by $\|\mathcal{P}''\|$, for $\mathcal{P}''$ the semi-grounding of $\mathcal{P} \cup \mathcal{D}$, while $\|\mathcal{P}''\|$ is easily seen to be polynomial in $\|\mathcal{D}\|$ for $\mathcal{P}$ fixed. Moreover, $\mathcal{P}''$ can be computed in polynomial time, w.r.t. $\|\mathcal{D}\|$, and each rule in $\mathcal{P}'$ can be computed from a rule in $\mathcal{P}''$ in polynomial time, provided that we can polynomially check $\mathcal{D}' \models \alpha$, for $\mathcal{D}'$ as above. This clearly holds since $\mathcal{D}' \models \alpha$ can be checked by simply matching α against facts in $\mathcal{D}'$. Finally $\max_{r \in \mathcal{P}'} \|r\|_u \leq \max_{r \in \mathcal{P}''} \|r\|_u \leq \max_{r \in \mathcal{P} \cup \mathcal{D}} \|r\|_u$. $\square$

We next use Lemmas 8 and 11 to show Lemma 12. To this end, we first establish the following auxiliary result.

Lemma A.8. *Let $\mathcal{P}$ be a semi-positive, limit-linear program and let f be the function mapping each triple $(\mathcal{D}, A, \vec{a})$, for $\mathcal{D}$ a limit dataset, A a* max *(resp.* min*) predicate and $\vec{a}$ a tuple of objects, to the greatest (resp. least) $k \in \mathbb{Z} \cup \{\infty\}$ such that $\mathcal{P} \cup \mathcal{D} \models A(\vec{a}, k)$ if such k exists, and otherwise to a special symbol* none. *Then function f is computable in* $\mathrm{FP}^{\mathrm{NP}}$.

Proof. Without loss of generality, suppose A is a max predicate. Let $\mathcal{P}'$ be the reduct of $\mathcal{P} \cup \mathcal{D}$, and let ℓ be the bound on the magnitude of integers in $\mathbf{P}(\mathcal{P}')$ from Lemma 8. Then, since, by Lemma 11, $\mathcal{P} \cup \mathcal{D} \models A(\vec{a}, k)$ implies $A(\vec{a}, k') \in \mathbf{P}(\mathcal{P}')$ for some $k' \geq k$, Lemma 8 implies that $\mathcal{P} \cup \mathcal{D} \models A(\vec{a}, \ell+1)$ if and only if $\mathcal{P} \cup \mathcal{D} \models A(\vec{a}, \infty)$. Similarly $\mathcal{P} \cup \mathcal{D} \not\models A(\vec{a}, -\ell)$ if and only if $\mathcal{P} \cup \mathcal{D}$ does not satisfy $A(\vec{a}, k)$ for any $k \in \mathbb{Z}$. Since $|\mathcal{P}'|$ is polynomial in $\|\mathcal{D}\|$ but $\max_{r \in \mathcal{P}' \setminus \mathcal{D}} \|r\|_u \leq \max_{r \in \mathcal{P}} \|r\|_u$, by Lemma 8, ℓ is exponentially bounded in $\|\mathcal{D}\|$, and hence every number in the range of f can be represented using polynomially many bits.

Given a triple $(\mathcal{D}, A, \vec{a})$, we can thus compute $f(\mathcal{D}, A, \vec{a})$ by a deterministic oracle TM whose oracle set consists of all pairs $(\mathcal{D}', \alpha)$ such that $\mathcal{P} \cup \mathcal{D}' \models \alpha$ as follows:

1. Compute the reduct $\mathcal{P}'$ of $\mathcal{P} \cup \mathcal{D}$.
2. Compute a bound ℓ on the magnitude of integers in $\mathbf{P}(\mathcal{P}')$ satisfying the restrictions in Lemma 8.
3. Perform a binary search for the greatest number $k \in [-\ell, \ell+1]$ such that $(\mathcal{D}, A(\vec{a}, k))$ is in the oracle set.
4. If no such k exists, return none, if $k = \ell + 1$, return ∞, and otherwise return k.

Correctness of the algorithm is immediate by the above observations.

The reduct can be computed in step (1) in polynomial time and is of polynomial size in $\|\mathcal{D}\|$, whereas $\max_{r \in \mathcal{P}'} \|r\|_u$ is bounded by a constant for a fixed $\mathcal{P}$ by Lemma 11. The computation in step (2) takes polynomial time as the binary representation of ℓ is polynomial in $\|\mathcal{D}\|$. The search in step (3) takes polynomial time and makes polynomially many oracle calls since the interval $[-\ell, \ell+1]$ is exponential in $\|\mathcal{D}\|$ and does not depend on A or $\vec{a}$, as observed above. Finally, step (4) is clearly polynomial in the size of the input.

The claim follows since, by the results in [Kaminski *et al.*, 2017], fact entailment for positive, limit-linear programs is CONP-complete, hence the membership problem for the oracle set is in CONP, and $\mathrm{FP}^{\mathrm{NP}} = \mathrm{FP}^{\mathrm{CONP}}$. ☐

We then generalise Lemma A.8 to Lemma 12.

Lemma 12. *Let $\mathcal{P}$ be a semi-positive, limit-linear program. Then the function mapping each limit dataset $\mathcal{D}$ to $\mathbf{P}(\mathcal{P} \cup \mathcal{D})$ is computable in* $\mathrm{FP}^{\mathrm{NP}}$ *in $\|\mathcal{D}\|$.*

Proof. The set $\mathbf{P}(\mathcal{P} \cup \mathcal{D})$ can be computed by the following algorithm:

1. Compute the reduct $\mathcal{P}'$ of $\mathcal{P} \cup \mathcal{D}$.
2. Compute the least (w.r.t. $\sqsubseteq$) pseudo-model J of all facts in $\mathcal{P}'$.
3. For each IDB predicate A and objects $\vec{a}$ occurring in the head of a rule in $\mathcal{P}'$:
   (a) if A is an object predicate and $\mathcal{P} \cup \mathcal{D} \models A(\vec{a})$, add $A(\vec{a})$ to J;
   (b) if A is a max (resp. min) predicate, compute the greatest (resp. least) $k \in \mathbb{Z} \cup \{\infty\}$ such that $\mathcal{P} \cup \mathcal{D} \models A(\vec{a}, k)$, and, if it exists, add $A(\vec{a}, k)$ to J.

Correctness of the algorithm follows since $\mathcal{P}'$ entails the same facts as $\mathcal{P} \cup \mathcal{D}$ by Lemma 11 and steps (2) and (3) construct the least pseudo-model of $\mathcal{P}'$. Thus, for the claim, it suffices to show that steps (1), (2), (3.a) and (3.b) are all feasible in $\mathrm{FP}^{\mathrm{NP}}$, while step (3) is repeated at most polynomially often in $\|\mathcal{D}\|$.

Step (1) can be performed in polynomial time in $\|\mathcal{D}\|$ by Lemma 11, while the construction of a pseudo-model of a dataset in step (2) is polynomial in $\|\mathcal{P}'\|$, and hence in $\|\mathcal{D}\|$, since it involves only trivial reasoning. Moreover, step (3) is repeated at most $|\mathcal{P}'|$ times, where $|\mathcal{P}'|$ is bounded polynomially in $\|\mathcal{D}\|$ for fixed $\mathcal{P}$. Finally, step (3.a) can be performed in CONP since fact entailment is CONP-complete in data complexity by the results in [Kaminski *et al.*, 2017], while step (3.b) is feasible in $\mathrm{FP}^{\mathrm{NP}}$ by Lemma A.8. ☐

Note that Lemma 12 immediately implies Corollary 9, so we dispense with a separate proof for the corollary.

Proposition 13. *If oracle O is computable in* FP^{C} *in data complexity, then Algorithm 1 runs in* P^{C} *in data complexity.*

Proof. Let $\mathcal{P} = \mathcal{P}_0 \cup \mathcal{D}$. Without loss of generality, the number of non-empty strata in $\mathcal{P}$ is bounded by a constant s for $\mathcal{P}_0$ fixed, and hence loop 3–5 is executed at most s times. Let J_i be the pseudo-interpretation computed by O in iteration i of the loop. By assumption, O in iteration i of the loop runs in time bounded by $q(\|\mathcal{D} \cup J_{i-1}\|)$, for some polynomial q, and hence $\|\mathcal{D} \cup J_i\| \leq p(\|\mathcal{D} \cup J_{i-1}\|)$ for some polynomial p. Consequently, we have $\|\mathbf{P}(\mathcal{P})\| = \|J_s\| \leq p(\|\mathcal{D}\|)^s$, which is in turn polynomial in $\|\mathcal{D}\|$, and loop 3–5 terminates in time bounded by $s \cdot q(p(\|\mathcal{D}\|)^s)$. Finally, step 6 can clearly be performed in time polynomial in $\|\mathbf{P}(\mathcal{P})\|$. ☐

Lemma 14. *For $\mathcal{P}$ a stratified, limit-linear program and α a fact, deciding $\mathcal{P} \models \alpha$ is in Δ_2^{P} in data complexity.*

Proof. The claim is immediate by Lemma 12 and Proposition 13. □

Theorem 15. *For $\mathcal{P}$ a stratified, limit-linear program and α a fact, deciding $\mathcal{P} \models \alpha$ is Δ_2^{P}-complete in data complexity. The lower bound holds already for programs with two strata.*

Proof. The upper bound follows by Lemma 14 while hardness is established by reduction from the *minimal satisfying assignment odd* problem. An instance $\mathcal{M}$ of the minimal satisfying assignment odd problem is given by a (repetition-free) tuple of variables $\langle x_N, \dots, x_0 \rangle$ and a satisfiable Boolean formula φ over $x_0, \dots, x_N$ (using operators $\vee$ and $\neg$). The problem is to determine whether the assignment σ for which the tuple $\langle \sigma(x_N), \dots, \sigma(x_0) \rangle$ is lexicographically minimal (assuming $\mathit{false} < \mathit{true}$) among all satisfying truth assignments of φ satisfies $\sigma(x_0) = \mathit{true}$. The closely related problem where $\langle \sigma(x_N), \dots, \sigma(x_0) \rangle$ is lexicographically maximal and φ is not restricted to be satisfiable has been shown Δ_2^{P}-complete in [Krentel, 1988, Theorem 3.4], and the two versions are easily seen to be LOGSPACE many-one inter-reducible. We reduce the problem by presenting a fixed program $\mathcal{P}_{modd}$, admitting two strata, and a dataset $\mathcal{D}_{\mathcal{M}}$, which depends on $\mathcal{M}$, and showing that $\mathcal{M}$ is true if and only if $\mathcal{P}_{modd} \cup \mathcal{D}_{\mathcal{M}}$ entails a nullary fact minOdd.

Our encoding uses object EDB predicates root, or, and not; ordinary numeric EDB predicate *shift*; and max IDB predicates ass, T, and F. Program $\mathcal{P}_{modd}$ consists of the following rules, where we write $(s_1 \leq s_2 < s_3)$ as an abbreviation for the conjunction $(s_1 \leq s_2) \wedge (s_2 < s_3)$.

$$\rightarrow \mathit{ass}(0) \tag{17}$$
$$\mathit{root}(x) \wedge F(x,n) \rightarrow \mathit{ass}(n+1) \tag{18}$$
$$\mathit{or}(x,y,z) \wedge F(y,n) \wedge F(z,n) \rightarrow F(x,n) \tag{19}$$
$$\mathit{or}(x,y,z) \wedge T(y,n) \rightarrow T(x,n) \tag{20}$$
$$\mathit{or}(x,y,z) \wedge T(z,n) \rightarrow T(x,n) \tag{21}$$
$$\mathit{not}(x,y) \wedge T(y,n) \rightarrow F(x,n) \tag{22}$$
$$\mathit{not}(x,y) \wedge F(y,n) \rightarrow T(x,n) \tag{23}$$
$$\mathit{ass}(n) \wedge \mathit{shift}(x,s) \wedge (0 \leq m_1) \wedge (0 \leq m_2 < s) \wedge (n \doteq 2 \times m_1 \times s + s + m_2) \rightarrow T(x,n) \tag{24}$$
$$\mathit{ass}(n) \wedge \mathit{shift}(x,s) \wedge (0 \leq m_1) \wedge (0 \leq m_2 < s) \wedge (n \doteq 2 \times m_1 \times s + m_2) \rightarrow F(x,n) \tag{25}$$
$$\lceil \mathit{ass}(n) \rceil \wedge (0 \leq m) \wedge (n \doteq 2 \times m + 1) \rightarrow \mathit{minOdd} \tag{26}$$

Dataset $\mathcal{D}_{(\vec{x},\varphi)}$ contains facts (27)–(30), where, for each distinct subformula ψ of φ (including φ itself), a_ψ is a fresh object. Note that numbers 2^i for $0 \leq i \leq N$ are exponential in N, and thus can be computed in polynomial time and represented using polynomially many bits in the size of the input.

$$\rightarrow \mathit{shift}(a_{x_i}, 2^i) \qquad \text{for each } 0 \leq i \leq N \tag{27}$$
$$\rightarrow \mathit{root}(a_\varphi) \tag{28}$$
$$\rightarrow \mathit{or}(a_\psi, a_{\psi_1}, a_{\psi_2}) \qquad \text{for each subformula } \psi = \psi_1 \vee \psi_2 \text{ of } \varphi \tag{29}$$
$$\rightarrow \mathit{not}(a_\psi, a_{\psi_1}) \qquad \text{for each subformula } \psi = \neg\psi_1 \text{ of } \varphi \tag{30}$$

In our reduction, each truth assignment σ for $x_0, \dots, x_N$ is associated with a number $\sum_{0 \leq i \leq N} \sigma(x_i) \times 2^i$. Thus, given a number n that encodes a truth assignment, variable x_i ($0 \leq i \leq N$) is assigned true if $n = 2 \times m_1 \times 2^i + 2^i + m_2$, and *false* if $n = 2 \times m_1 \times 2^i + m_2$, for some nonnegative integers m_1 and m_2 where $m_2 < 2^i$. Thus, if numeric variable n is assigned such an encoding of a truth assignment and numeric variable s is assigned the factor 2^i corresponding to variable x_i, then conjunction

$$(0 \leq m_1) \wedge (0 \leq m_2 < s) \wedge (n \doteq 2 \times m_1 \times s + s + m_2)$$

is true if and only if x_i is true in the assignment (encoded by) n; analogously, conjunction

$$(0 \leq m_1) \wedge (0 \leq m_2 < s) \wedge (n \doteq 2 \times m_1 \times s + m_2)$$

is true if and only if x_i is *false* in assignment n. Facts (27) associate with every variable x_i the corresponding factor 2^i, and hence rules (24) and (25) derive $T(a_{x_i}, n)$ if x_i is true and $F(a_{x_i}, n)$ if x_i is *false* in assignment n. Facts (28)–(30) encode the structure of φ. Using these facts, rules (19)–(23) recursively evaluate φ, deriving, for each subformula ψ of φ, $T(a_\psi, n)$ if ψ evaluates to true and $F(a_\psi, n)$ if ψ evaluates to *false* in assignment n. Rules (17) and (18) then search for the lexicographically minimal assignment that satisfies φ (recall that φ is satisfiable by assumption)—rule (17) ensures than assignment 0 is checked, and rule (18) ensures that assignment $n+1$ is checked whenever φ evaluates to *false* in n. Finally, rule (26) derives minOdd if and only if x_0 is true in the minimal assignment satisfying φ, as required. □

B Proofs for Section 5

Lemma 17. *For $\mathcal{P}$ a semi-positive, type-consistent program and $\mathcal{D}$ a limit dataset, the reduct of $\mathcal{P} \cup \mathcal{D}$ is polynomially rewritable to a positive, semi-ground, type-consistent program $\mathcal{P}'$ such that, for each fact α, $\mathcal{P} \cup \mathcal{D} \models \alpha$ if and only if $\mathcal{P}' \models \alpha$.*

Proof. By definition, the program $\mathcal{P}'$ obtained by first semi-grounding $\mathcal{P} \cup \mathcal{D}$ and then simplifying all numeric terms as much as possible is type-consistent. Thus, it suffices to show that the possible violations of type consistency introduced by the additional transformation rules in Definition 10 can be repaired in polynomial time. Since the transformation rules apply to negative body literals of an individual rule, suppose r is a semi-ground, semi-positive, type-consistent rule and $\mu = \mathsf{not}\,\alpha$ is a negative body literal of r. We have two cases.

If α is ground, then either r is deleted or μ is deleted from r. Clearly, neither of these transformations can violate type consistency since μ does not mention a numeric variable.

If $\alpha = A(\vec{a}, m)$ is a non-ground limit literal, then one of the following is true, for $\mathcal{D}'$ the set of facts in $\mathcal{P} \cup \mathcal{D}$:

(i) $\mathcal{D}' \models A(\vec{a}, k)$ for each $k \in \mathbb{Z}$ and r is removed,
(ii) $\mathcal{D}' \not\models A(\vec{a}, k)$ for each $k \in \mathbb{Z}$ and μ is removed from r, or
(iii) $\mathcal{D}' \models \lceil A(\vec{a}, k) \rceil$ for some $k \in \mathbb{Z}$ and μ is replaced in r with $(k \prec_A m)$.

Case *(i)* does not violate type consistency.

In case *(ii)*, since r is type-consistent, variable m needs to be guarded, i.e., there needs to be a conjunction $A(\vec{a}, n) \wedge (m \doteq n + t)$ for $t \in \{1, -1\}$ in $\mathsf{b}(r)$. Moreover, since A is EDB in $\mathcal{P} \cup \mathcal{D}$, and hence in $\mathcal{P}'$, $\mathcal{D}' \not\models A(\vec{a}, k)$ implies $\mathcal{P}' \not\models A(\vec{a}, k)$ for each $k \in \mathbb{Z}$, and thus the literal $A(\vec{a}, n) \in \mathsf{b}(r)$ will never be satisfied when computing the materialisation of $\mathcal{P}'$, i.e., r is semantically redundant and hence can be removed from $\mathcal{P}'$, maintaining type-consistency.

Finally, in case *(iii)*, type consistency is violated because in the transformed rule, variable m no longer occurs in a standard body atom. Let r' be the rule obtained from r by a one-step application of the transformation rules in case *(iii)*. To restore type consistency, we will equivalently re-state rule r' eliminating all occurrences of m. To this end, note that, as in the previous case, since r is type-consistent, variable m needs to be guarded, i.e., there needs to be a conjunction $A(\vec{A}, n) \wedge \mathsf{not}\,A(\vec{a}, m) \wedge (m \doteq n + t)$ for $t \in \{1, -1\}$ in $\mathsf{b}(r)$. Thus, let rule r'' be obtained from r by removing the conjunction $A(\vec{a}, n) \wedge \mathsf{not}\,A(\vec{a}, m) \wedge (m \doteq n + t)$ and substituting each occurrence of m in r with $k + t$ and each occurrence of n with k. Clearly, rule r'' is type-consistent since so is r. Moreover, since A is EDB in $\mathcal{P}'$, we have $\mathcal{P}' \models \lceil A(\vec{a}, k) \rceil$, and hence r can be replaced with r'' while maintaining the set of entailed facts.

Clearly, the transformation rules in Definition 10 restricted to type-consistent programs can be modified to preserve type consistency as described above while remaining polynomially computable. $\square$

Theorem 18. *For $\mathcal{P}$ a stratified, type-consistent program and α a fact, deciding $\mathcal{P} \models \alpha$ is* P*-complete in data complexity.*

Proof. The P lower bound in data complexity is inherited from plain Datalog [Dantsin *et al.*, 2001]. For the upper bound, note that, for $\mathcal{P}$ a stratified, type-consistent program, the program $\mathcal{P}[i] \cup \mathcal{D}$ is type-consistent for each $i \in \mathbb{N}$ and each finite dataset $\mathcal{D}$. Thus, by Proposition 13, it suffices to show that the pseudo-materialisation of a semi-positive, type-consistent program $\mathcal{P}'$ can be computed in polynomial time in data complexity. By Lemmas 11 and 17, this reduces to showing the existence of a polynomial algorithm for computing the pseudo-materialisation of a semi-ground, positive, type-consistent program $\mathcal{P}''$. Kaminski *et al.* [2017] provide such an algorithm that terminates in time polynomial in $\|\mathcal{P}''\|$, provided $\max_{r \in \mathcal{P}''} \|r\|_u$ is bounded by a constant. This assumption can be made since, w.l.o.g., $\max_{r \in \mathcal{P}'} \|r\|_u$ is constant w.r.t. data complexity and, for $\mathcal{P}''$ the semi-ground, positive, type-consistent program obtained from $\mathcal{P}'$ by the results in Lemmas 11 and 17, we have $\max_{r \in \mathcal{P}''} \|r\|_u \leq \max_{r \in \mathcal{P}'} \|r\|_u$. $\square$

Proposition 19. *Checking whether a stratified, limit-linear program is type-consistent is in* LogSpace.

Proof. Let $\mathcal{P}$ be a stratified, limit-linear program. We can check whether $\mathcal{P}$ is type-consistent by considering each rule $r \in \mathcal{P}$ independently. For the first type consistency condition, note that each maximally simplified numeric term in a semi-ground limit-linear rule has the form $k_0 + \sum_{i=1}^{n} k_i \times \prod_{j=1}^{\ell_i} m_i^j$ for all $\ell_i \geq 1$. Such a term satisfies the first condition iff, for each i, either $\ell_i = 1$ or $k_i = 0$. Thus, to check the first condition, it suffices, for each numeric term $s_0 + \sum_{i=1}^{n} s_i$ in r and each i such that s_i contains at least two variables not occurring in a positive ordinary numeric literal, to check that either one of the constants in s_i is 0 or s_i contains some variable m occurring in a positive ordinary numeric literal and 0 is the only constant mentioned in $\mathcal{P}$ (and hence m must be semi-grounded to 0); clearly, this is doable in logarithmic space.

The second and third conditions are clearly checkable in logarithmic space.

Thus, it suffices to check whether a semi-grounding of r (with constants from $\mathcal{P}$) can violate the fourth or the fifth condition. In both cases, it suffices to consider at most one atom α at a time (a limit head atom $A(\mathbf{a}, s)$ for the fourth condition or a comparison atom $s_1 < s_2$ or $s_1 \leq s_2$ for the fifth condition). In α, we consider at most one numeric term s at a time ($s \in \{s_1, s_2\}$ for the fifth condition), where, by our considerations for the first condition, we can assume w.l.o.g. that s has the form $t_0 + \sum_{i=1}^{n} t_i \times m_i$ where t_i, for $i \geq 1$, are terms constructed from integers, variables occurring in positive ordinary numeric literals, and multiplication. Moreover, for each such s, we consider each unguarded variable m occurring in s. By assumption, m occurs in s, so we have $m_i = m$ for some i. For the fourth condition of Definition 16, we need to check that, if the positive limit body literal $B(\mathbf{s}, m_i)$ introducing m_i (note that m_i cannot be introduced by a negative literal by the third condition and since it is unguarded by assumption) has the same (different) type as the head atom, then term t_i can only be grounded to positive (negative) integers or zero. For the fifth condition, we need to check that, if $s = s_1$ and the positive limit body literal $B(\mathbf{s}, m_i)$ introducing m_i is min (max), then term t_i can only be grounded to positive (negative) integers or 0, and dually for the case $s = s_2$. Hence, in either case, it suffices to check whether term t_i can be semi-grounded so that it evaluates to a positive integer, a negative integer, or zero. We next discuss how this can be checked in logarithmic space. Let $t_i = t_i^1 \times \cdots \times t_i^k$, where each t_i^j is an integer or a variable not occurring in a limit atom, and assume without loss of generality that we want to check whether t_i can be grounded to a positive integer; this is the case if and only if one of the following holds:

- all t_i^j are integers whose product is positive;
- the product of all integers in t_i is positive and $\mathcal{P}$ contains a positive integer;
- the product of all integers in t_i is positive, $\mathcal{P}$ contains a negative integer, and the total number of variable occurrences in t_i is even;
- the product of all integers in t_i is negative, $\mathcal{P}$ contains a negative integer, and the total number of variable occurrences in t_i is odd; or
- the product of all integers in t_i is negative, $\mathcal{P}$ contains both positive and negative integers, and some variable t_i^j has an odd number of occurrences in t_i.

Each of these conditions can be verified using a constant number of pointers into $\mathcal{P}$ and binary variables. This clearly requires logarithmic space, and it implies our claim. $\square$