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; Gangan 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.

\[
\text{node}(x) \rightarrow d(x, x, 0) \tag{1}
\]
\[
d(x, y, m) \land \text{edge}(y, z, n) \rightarrow d(x, z, m + n) \tag{2}
\]

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:

\[
d(x, y, k) \land (k \leq k') \rightarrow d(x, y, k'). \tag{3}
\]

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 \text{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 \( \Delta^2 \)-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 an extended version of this paper available online at arXiv:1804.09473.

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 \(+, -, \times\) are the standard arithmetic functions. A constant is an object or an integer. A standard atom is of the form \( B(t_1, \ldots, t_n) \), 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 \( \neg \alpha \) for \( \alpha \) a standard atom. A comparison atom is of the form \((s_1 < s_2)\) or \((s_1 \leq s_2)\), with \( \leq \) and \( \leq \) the usual comparison predicates over the integers, and \( s_1 \) and \( s_2 \) numeric terms. We write \((s_1 \equiv s_2)\) as an abbreviation for \((s_1 \leq s_2) \land (s_2 \leq s_1)\). A term, atom or literal is ground if it has no variables.

A rule \( r \) has the form \( \bigwedge_i \mu_i \land \bigwedge_j \beta_j \rightarrow \alpha \), where the body \( \bigwedge_i \mu_i \land \bigwedge_j \beta_j \) is a possibly empty conjunction of standard literals \( \mu_i \) and comparison atoms \( \beta_j \), and the head \( \alpha \) 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 \((x = s)\) 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 \( \rightarrow \) and use symbol \( \infty \) instead of variables when writing facts. A dataset \( D \) is a finite set of facts. Dataset \( D \) is ordered if \( (i) \) it contains facts \( \text{first}(a_1), \text{next}(a_1, a_2), \ldots, \text{next}(a_{n-1}, a_n), \text{last}(a_n) \) for some repetition-free enumeration \( a_1, \ldots, a_n \) of all objects in \( D \); and \( (ii) \) it contains no other facts over predicates \( \text{first, next, and last} \). 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 \( P \) if \( B \) occurs in \( P \) in the head of a rule that is not a fact; otherwise, \( B \) is extensional (EDB) in \( P \). Program \( P \) 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 \( P \) is a function \( \lambda \) mapping each predicate to a positive integer such that, for each rule with the head over a predicate \( A \) and each standard body literal \( \mu \) over \( B \), we have \( \lambda(B) \leq \lambda(A) \) if \( \mu \) is positive, and \( \lambda(B) < \lambda(A) \) if \( \mu \) is negative. Program \( P \) is stratified if it admits a stratification. Given a stratification \( \lambda \), we write \( P[i] \) for the \( i \)-th stratum of \( P \) over \( \lambda \)—that is, the set of all rules in \( P \) 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 \( \infty \)). Interpretation \( I \) satisfies a ground atom \( \alpha \), written \( I \models \alpha \), if either \( (i) \alpha \) is a standard atom such that evaluation of the arithmetic functions in \( \alpha \) under the usual semantics over integers produces a fact in \( I \); or \( (ii) \alpha \) is a comparison atom that evaluates to \( \text{true} \) under the usual semantics. Interpretation \( I \) satisfies a ground negative literal \( \neg \alpha \), written \( I \models \neg \alpha \), 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 \( P \), then \( I \) is a model of \( P \). For \( I \) a Herbrand interpretation and \( R \) a (possibly infinite) semi-positive set of rules, let \( S_R(I) \) be the set of facts \( \alpha \) such that \( \varphi \rightarrow \alpha \) is a ground instance of a rule in \( R \) and \( I \models \varphi \). Given a program \( P \) and a stratification \( \lambda \) of \( P \), for each \( i, j \geq 0 \) we define interpretation \( I^i_{j+1} \) by induction on \( i \) and \( j \):

\[
I^0_0 = I^0_1 = \emptyset; \quad I^i_{j+1} = S^i_{P[j+1]} \cup I^\infty_{j+1}; \quad I^\infty_i = \bigcup_{j \geq 0} I^i_j.
\]

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

Reasoning We study the computational properties of checking whether \( P \cup D \models \alpha \), for \( P \) a program, \( D \) a dataset, and \( \alpha \) a fact. We are interested in data complexity, which assumes
that only $D$ and $\alpha$ form the input while $P$ is fixed. Unless otherwise stated, all numbers in the input are coded in binary, and the size $|P|$ of $P$ is the size of its representation. Checking $P \cup D \models \alpha$ is undecidable even if the only arithmetic function in $P$ 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, $\Delta_2^P$ 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 $(P, \tau)$ where
- $P$ 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
- $\tau$ is a partial function from numeric predicates to $\{\min, \max\}$ that is total on the IDB predicates in $P$ and on predicates occurring in non-ground facts.

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

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

Intuitively, a limit fact $B(\bar{a}, k)$ says that the value of $B$ for a tuple of objects $\bar{a}$ 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 $(P, \tau)$ if it satisfies the program $P \cup ax(P)$, where $ax(P)$ contains the following rule for each limit predicate $A$ in $P$:

$$A(\bar{x}, m) \land (n \geq_A m) \rightarrow A(\bar{x}, n).$$

The materialisation $M(P, \tau)$ of $(P, \tau)$ is $M(P \cup ax(P))$; and $(P, \tau)$ entails $\alpha$, written $(P, \tau) \models \alpha$, if $\alpha \in M(P, \tau)$.

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 $[A(\bar{x}, n)]$ of a max (or min) limit atom $A(\bar{x}, n)$ is the conjunction $A(\bar{x}, n) \land \neg A(\bar{x}, m) \land (m = n + t)$ where $t = 1$ (or $t = -1$, respectively) and $m$ is a fresh variable.

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

**Example 4.** An input of the single-pair shortest path problem can be encoded in the obvious way as a dataset $D_{sp}$ 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 $P_{sp}$ given next computes, together with $D_{sp}$ (where all edge weights are positive), a DAG over a binary object predicate $sp-edge$ such that every maximal path in the DAG is a shortest path from $u$ to $v$.

$$source(x) \rightarrow ds(x, 0)$$

$$ds(x, m) \land edge(x, y, n) \rightarrow ds(x, m + n)$$

$$(ds(x, m_1) \land ds(y, m_2)) \land edge(x, y, n) \land target(y) \land (m_1 + n = m_2) \rightarrow sp-edge(x, y)$$

$$(ds(x, m_1) \land ds(y, m_2)) \land edge(x, y, n) \land sp-edge(y, z) \land (m_1 + n = m_2) \rightarrow sp-edge(x, y)$$

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, $P_{sp} \cup D_{sp} \models ds(v, k)$ 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 $sp-edge$ predicate such that $P_{sp} \cup D_{sp} \models sp-edge(a, b)$ if and only if the edge $(a, b)$ is part of a shortest path from $u$ to $v$.

**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 $1/\sum_{v \in \text{node in } G} d(u, v)$, 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 $D_{cs}$ using, as before, a unary object predicate *node* and a ternary ordinary numeric predicate *edge*. Program $P_{cs}$ consists of rules (8)–(16), where $d$, *farness* and *farness* are min predicates, and *centre* and *centre* are object predicates.

$$node(x) \rightarrow d(x, x, 0)$$

$$d(x, y, m) \land edge(y, z, n) \rightarrow d(x, z, m + n)$$

$$first(y) \land d(x, y, n) \rightarrow farness(x, y, n)$$

$$next(y, z) \land farness(x, y, m) \land d(x, z, n) \rightarrow farness(x, z, m + n)$$

$$farness(x, y, n) \land last(y) \rightarrow farness(x, n)$$
first(x) → centre'(x, x)  \tag{13}

next(x, y) \land centre'(x, z) \land \[fness(z, n)] \land \[fness(y, m)] \land (m < n) \rightarrow centre'(y, y)  \tag{14}

next(x, y) \land centre'(x, z) \land \[fness(z, n)] \land \[fness(y, m)] \land (n < m) \rightarrow centre'(y, z)  \tag{15}

centre'(x, z) \land 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.

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 \(\Delta^2_n\)-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

\[ A(s, n_1), \text{ not } A(s, n_2), \quad (n_2 = n_1 + t), \]

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 \(P\) is obtained by replacing, in every rule \(r\) in \(P\), each object variable and each numeric variable occurring in an ordinary numeric atom in \(r\) with a constant in \(P\) in all possible ways.

It is easily seen that the semi-grounding of a limit-linear program \(P\) entails the same facts as \(P\) 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.

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

\(\text{Step 2.}\) 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 \(\text{FP}^\text{NP}\) oracle \(O\) for computing the pseudo-materialisation of a semi-positive limit-linear program.

\(\text{Step 3.}\) We provide an algorithm (see Algorithm 1) that decides entailment of a fact \(\alpha\) by a stratified limit-linear program \(P\) 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 \(\alpha\) is obtained, is computed using a constant number of oracle calls in the size of the data, which yields a \(\Delta^2_n\) 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.
\( \mathcal{P}(\mathcal{P}) \) of a stratified limit program \( \mathcal{P} \), which compactly represents the materialisation \( \mathcal{M}(\mathcal{P}) \). Intuitively, \( \mathcal{M}(\mathcal{P}) \) can be infinite because it can contain, for any limit predicate \( \mathcal{B} \) and tuple of objects \( \mathcal{a} \) of suitable arity, an infinite number of facts of the form \( \mathcal{B}(\mathcal{a}, k) \). However, if the materialisation has facts of this form, then either there is a limit value \( \ell \) such that \( \mathcal{B}(\mathcal{a}, k) \in \mathcal{M}(\mathcal{P}) \) for each \( k \leq \ell \) and \( \mathcal{B}(\mathcal{a}, k') \notin \mathcal{M}(\mathcal{P}) \) for each \( k' > \ell \), or \( \mathcal{B}(\mathcal{a}, k) \in \mathcal{M}(\mathcal{P}) \) for every integer \( k \). As argued in prior work, it then suffices for the pseudo-materialisation to contain only a single fact \( \mathcal{B}(\mathcal{a}, \ell) \) in the former case, or \( \mathcal{B}(\mathcal{a}, \infty) \) in the latter case.

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

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

1. an object or ordinary numeric fact is contained in \( \mathcal{P}(\mathcal{P}) \) if and only if it is contained in \( \mathcal{M}(\mathcal{P}) \); and
2. for each limit predicate \( \mathcal{B} \), object tuple \( \mathcal{a} \), and integer \( \ell \),
   - \( \mathcal{B}(\mathcal{a}, \ell) \in \mathcal{P}(\mathcal{P}) \) if and only if \( \mathcal{B}(\mathcal{a}, \ell) \in \mathcal{M}(\mathcal{P}) \) and \( \mathcal{B}(\mathcal{a}, k) \notin \mathcal{M}(\mathcal{P}) \) for all \( k > \ell \), and
   - \( \mathcal{B}(\mathcal{a}, \infty) \in \mathcal{P}(\mathcal{P}) \) if and only if \( \mathcal{B}(\mathcal{a}, k) \in \mathcal{M}(\mathcal{P}) \) 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 \( |\mathcal{P}(\mathcal{P} \cup \mathcal{D})| \leq |\mathcal{P} \cup \mathcal{D}| \) and the magnitude of each integer in \( \mathcal{P}(\mathcal{P} \cup \mathcal{D}) \) 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 \( \mathcal{P} \cap \mathcal{D} \) 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 \( \mathsf{FP}^\mathsf{NP} \) 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 \( \mathcal{P}(\mathcal{P} \cup \mathcal{D}) \) is computable in \( \mathsf{FP}^\mathsf{NP} \) 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 \( \mu \), it suffices to check whether \( \mu \) is entailed by the facts in \( \mathcal{P} \) and simplify all rules containing \( \mu \) 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 \( \mu \) in \( r \):

1. if \( \mu = \text{not } \alpha \), for a ground atom, delete \( r \) if \( \mathcal{D} \models \alpha \), and delete \( \mu \) from \( r \) otherwise.
2. if \( \mu = \text{not } \mathcal{A}(\mathcal{a}, m) \) is a non-ground limit literal, then
   - delete \( r \) if \( \mathcal{D} \models \mathcal{A}(\mathcal{a}, k) \) for each integer \( k \);
   - delete \( \mu \) from \( r \) if \( \mathcal{D} \not\models \mathcal{A}(\mathcal{a}, k) \) for each \( k \); and
   - replace \( \mu \) in \( r \) with \( (k < A_m) \) otherwise, where \( \mathcal{D} \models [\mathcal{A}(\mathcal{a}, k)] \).

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 \( \alpha \) a fact, we have \( \mathcal{P} \cup \mathcal{D} \models \alpha \) if and only if \( \mathcal{P}' \cup \mathcal{D} \models \alpha \). 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 \( \mathsf{FP}^\mathsf{NP} \) 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 \( \mathcal{P}(\mathcal{P} \cup \mathcal{D}) \) is computable in \( \mathsf{FP}^\mathsf{NP} \) in \( |\mathcal{D}| \).

We are now ready to present Algorithm 1, which decides entailment of a fact \( \alpha \) by a stratified limit-linear program \( \mathcal{P} \). The algorithm uses an oracle \( \mathcal{O} \) 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 \( \mathcal{P}(\mathcal{P}) \) of \( \mathcal{P} \) stratum by stratum in a bottom-up fashion. For each stratum \( i \), the algorithm uses oracle \( \mathcal{O} \) 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 \( \mathcal{P}(\mathcal{P}) \) has been constructed, entailment of \( \alpha \) is checked directly over \( \mathcal{P}(\mathcal{P}) \).

Correctness of the algorithm is immediate by the properties of \( \mathcal{O} \) and the correspondence between pseudo-materialisations and materialisations. Moreover, if oracle \( \mathcal{O} \) runs in \( \mathsf{FP}^\mathsf{NP} \) 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 $\mathbb{P}^{\mathbb{C}}$ since the number of strata of $\mathcal{P}$ does not depend on the input dataset.

**Proposition 13.** If oracle $O$ is computable in $\mathbb{FP}^{\mathbb{C}}$ in data complexity, then Algorithm 1 runs in $\mathbb{P}^{\mathbb{C}}$ 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 $\alpha$ a fact, deciding $\mathcal{P} \models \alpha$ is in $\Delta_2^p$ 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 $\varphi$ over these variables. The question is whether the truth assignment $\sigma$ satisfying $\varphi$ for which the tuple $\langle \sigma(x_N), \ldots, \sigma(x_0) \rangle$ is lexicographically minimal, assuming $\text{false} < \text{true}$, among all satisfying truth assignments of $\varphi$ has $\sigma(x_0) = \text{true}$. In our reduction, $\mathcal{M}$ is encoded as a dataset $\mathcal{D}_M$ using object predicates or and not to encode the structure of $\varphi$ and numeric predicates to encode the order of variables in $\langle x_N, \ldots, x_0 \rangle$; a fixed, two-strata program $\mathcal{P}_{\text{modd}}$ then goes through all assignments $\sigma$ in the ascending lexicographic order and evaluates the encoding of $\varphi$ on $\sigma$ until it finds some $\sigma$ that makes $\varphi$; $\mathcal{P}_{\text{modd}}$ then derives fact $\text{minOdd}$ if and only if $\sigma(x_0) = \text{true}$. Thus, $\mathcal{P}_{\text{modd}} \cup \mathcal{D}_M \models \text{minOdd}$ if and only if $\mathcal{M}$ belongs to the language of ODDMINSAT.

**Theorem 15.** For $\mathcal{P}$ a stratified, limit-linear program and $\alpha$ a fact, deciding $\mathcal{P} \models \alpha$ is $\Delta_2^p$-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, stratified, 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 $A(x, s)$ 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 $\infty$ 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-linear, type-consistent program $\mathcal{P}'$ such that, for each fact $\alpha$, $\mathcal{P} \cup \mathcal{D} \models \alpha$ 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 $\alpha$ 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 $\text{LOGSPACE}$.

### 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 $\Delta_2^p$. 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 computa-
tion 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. Al-
though 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 investi-
gate the theoretical properties of limit Datalog under well-
founded [Van Gelder et al., 1991] or the stable model seman-
tics [Gelfond and Lifschitz, 1988]. Finally, we plan to imple-
ment our reasoning algorithms and test them in practice.

Acknowledgments

This research was supported by the EPSRC projects DBOnto,
MaSI3, and ED3.

References

[Alvaro et al., 2010] Peter Alvaro, Tyson Condie, Neil Con-
way, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell
Sears. BOOM analytics: exploring data-centric, declarative

[Beeri et al., 1991] Catriel Beeri, Shamir A. Naqvi, Oded
Shmueli, and Shalom Tsur. Set constructors in a logic

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

[Consens and Mendelson, 1993] Mariano P. Consens and
Alberto O. Mendelson. Low complexity aggregation in

[Dantsin et al., 2001] Evgeny Dantsin, Thomas Eiter, Georg
Gottlob, and Andrei Voronkov. Complexity and expressive
power of logic programming. ACM Comput. Surv.,

[Eisner and Filaro, 2011] Jason Eisner and Nathaniel Wes-
ley Filaro. Dyna: Extending datalog for modern AI.

[Ganguly et al., 1995] Sumit Ganguly, Sergio Greco, and
Carlo Zaniolo. Extrema predicates in deductive databases.

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

[Kaminski et al., 2017] Mark Kaminski, Bernardo Cuenca
Grau, Egor V. Kostylev, Boris Motik, and Ian Horrocks.
Foundations of declarative data analysis using limit data-
log 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

[Krentel, 1988] Mark W. Krentel. The complexity of opti-

[Loo et al., 2009] Boon Thau Loo, Tyson Condie, Minos N.
Garofalakis, David E. Gay, Joseph M. Hellerstein, Pet-
ros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and
Ion Stoica. Declarative networking. Commun. ACM,

declarative data analysis and data independence in the big

[Mazuran et al., 2013] Mirjana Mazuran, Edoardo Serra,
and Carlo Zaniolo. Extending the power of datalog recur-

[Munick et al., 1990] Inderpal Singh Munick, Hamid Pir-
hesh, and Raghu Ramakrishnan. The magic of duplicates
Kaufmann, 1990.

[Ross and Sagiv, 1997] Kenneth A. Ross and Yehoshua Sa-
giv. Monotonic aggregation in deductive databases.

[Sabidussi, 1966] Gert Sabidussi. The centrality index of a

[Seo et al., 2015] Jiwon Seo, Stephen Guo, and Monica S.
Lam. SociaLite: An efficient graph query language based
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.

Ross, and John S. Schlipf. The well-founded semantics for


[Wang et al., 2015] Jingjing Wang, Magdalena Balazinska,
and Daniel Halperin. Asynchronous and fault-tolerant
recursive datalog evaluation in shared-nothing engines.