Limit Datalog: A Declarative Query Language for Data Analysis

Bernardo Cuenca Grau
University of Oxford

Ian Horrocks
University of Oxford

Mark Kaminski
University of Oxford

Egor V. Kostylev
University of Oxford

Boris Motik
University of Oxford

ABSTRACT

Motivated by applications in declarative data analysis, we study $Datalog_{\mathbb{Z}}$—an extension of Datalog with stratified negation and arithmetics over integers. Reasoning in this language is undecidable, so we present a fragment, called limit $Datalog_{\mathbb{Z}}$, that is powerful enough to naturally capture many important data analysis tasks. In limit $Datalog_{\mathbb{Z}}$, all intensional predicates with a numeric argument are *limit predicates* that keep only the maximal or minimal bounds on numeric values. Reasoning in limit $Datalog_{\mathbb{Z}}$ is decidable if multiplication is used in a way that satisfies our *linearity condition*. Moreover, fact entailment in limit-linear $Datalog_{\mathbb{Z}}$ is Δ_2^{EXP}-complete in combined and Δ_2^{P}-complete in data complexity, and it drops to coNEXP and coNP, respectively, if only (semi-)positive programs are considered. We also propose an additional stability requirement, for which the complexity drops to EXP and P, matching the bounds for usual Datalog. Limit $Datalog_{\mathbb{Z}}$ thus provides us with a unified logical framework for declarative data analysis and can be used as a basis for understanding the expressive power of the key data analysis constructs.

1. INTRODUCTION

Analysing complex datasets is a hot topic in information systems. The term 'data analysis' covers a broad range of tasks such as data aggregation, property verification, and query answering. Such tasks are usually realised in practice using imperative programming languages. However, there has recently been considerable interest in declarative solutions, where the definition of the task is clearly separated from its implementation [1, 18, 25, 26, 29]. The main idea is that users should describe *what* the desired output is, rather than *how* to compute it. For example, instead of computing shortest paths in a graph using a concrete algorithm, one first describes what a path length is and then selects the paths of minimum length. Such a specification is independent of evaluation details, allowing analysts to focus on their task at hand. An evaluation strategy can be selected independently, typically by reusing efficient general algorithms 'for free'.

An essential ingredient of declarative data analysis is a suitable logic-based language for representing the relevant analysis tasks. Datalog [8] is a prime candidate due to its support for recursion, which is needed to express common analysis problems such as shortest path. However, in addition to standard features of Datalog, even basic data analysis tasks often require integer arithmetic or aggregation to capture quantitative aspects of data (e.g., the length of a shortest path). Research on extending recursive rule languages with means for capturing numeric computations dates back to the '90s [3, 6, 12, 16, 20, 22, 28], and is currently experiencing a revival [11, 19, 31]. This large body of work, however, focuses primarily on integrating recursion with arithmetic and aggregation in a coherent semantic framework, where technical difficulties arise due to nonmonotonicity. Surprisingly, little is known about the computational properties of languages integrating recursion with arithmetic, other than the fact that a straightforward combination is undecidable [8]. This result applies also to the languages of existing Datalog-based tools such as BOOM [1], DeALS [30], LogicBlox [2], Myria [29], SociaLite [25], Overlog [17], Dyna [9], and Yedalog [4].

The aim of this article is to lay the foundation for Datalog-based declarative data analysis by developing new extensions of Datalog that are powerful and flexible enough to naturally capture many important analysis tasks, and that yet exhibit favourable computational properties of reasoning. These languages can provide a formal basis for the development of reasoning engines that support complex analytical tasks and provide correctness, robustness, scalability, and extensibility guarantees. They can

also serve as a unified logical framework providing a basis for understanding the expressive power of the key data analysis constructs.

We take as a starting point $Datalog_{\mathbb{Z}}$—a well-known extension of Datalog with stratified negation as failure, integer arithmetic, and comparisons. After reviewing basic definitions in Section 2, in Section 3.1 we present *limit* $Datalog_{\mathbb{Z}}$, which can be equivalently seen as either a semantic or a syntactic restriction of $Datalog_{\mathbb{Z}}$. In limit $Datalog_{\mathbb{Z}}$, all intensional predicates with numeric arguments are *limit* predicates that keep maximal or minimal bounds on the numeric value for each tuple of the other arguments. For example, if we encode a directed graph with edge lengths using a ternary predicate E, then Rules (1) and (2), where dst is a min limit predicate, compute the length of a shortest path from a source node a_s to each node in the graph.

$$\rightarrow dst(a_s, 0) \tag{1}$$

$$dst(x, m) \wedge E(x, y, n) \rightarrow dst(y, m + n) \tag{2}$$

Rule (2) intuitively says that, if x is reachable from a_s with length at most m and (x, y) is an edge of length n, then y is reachable from a_s with length at most $m + n$. If these rules and a dataset entail $dst(a, \ell)$, then the length of a shortest path from a_s to a is at most ℓ; thus, $dst(a, k)$ holds for each $k \geq \ell$ since the length of a shortest path is also at most k. This is different from ordinary $Datalog_{\mathbb{Z}}$, where $dst(a, \ell)$ and $dst(a, k)$ are not semantically connected. In Section 3.2, we show that fact entailment remains undecidable for limit $Datalog_{\mathbb{Z}}$ programs. To ensure decidability, we introduce *limit-linear* $Datalog_{\mathbb{Z}}$, which disallows multiplication of numeric variables that are used in the same stratum. In Section 3.3, we present several examples that show how limit-linear $Datalog_{\mathbb{Z}}$ can capture many practically relevant data analysis tasks.

In Section 4, we establish decidability of fact entailment for limit-linear $Datalog_{\mathbb{Z}}$ and design worst-case optimal algorithms for positive (i.e., negation-free), semi-positive (i.e., with negation only in front of extensional atoms), and arbitrary (i.e., with stratified negation) limit-linear programs. Our results are obtained by a reduction to the evaluation problem for sentences of a specific shape in Presburger arithmetic. In particular, in Section 4.1 we first design a fact entailment algorithm for positive limit-linear programs with coNEXP and coNP upper complexity bounds in combined and data complexity, respectively, and then show that these bounds are worst-case optimal. In Section 4.2, we first show that fact entailment for semi-positive programs can be reduced in polynomial time to the positive case and then design a fact entailment algorithm for arbitrary limit-linear programs that materialises the input stratum by stratum, by relying at each stage on an oracle computing the materialisation of a semi-positive program corresponding to the previous strata. This algorithm provides Δ_2^{EXP} and Δ_2^{P} upper complexity bounds, and we show that these bounds are also worst-case optimal.

The results of Section 4 establish intractability of reasoning over limit-linear programs. In Section 5, we identify fragments of our language for which reasoning is tractable in data complexity, and which are therefore well-suited for data-intensive applications. In particular, using the idea of *cyclic dependency* detection, in Section 5.1 we introduce *stable* programs that allow reasoning to become EXP-complete in combined complexity and P-complete in data complexity (i.e., no harder than for ordinary Datalog). Stability, however, is a semantic condition that is hard to check; thus, in Section 5.2, we identify a syntactic *type-consistency* condition, which implies stability and can be easily checked rule by rule. We then argue that all analysis tasks discussed in our examples can be captured using type-consistent $Datalog_{\mathbb{Z}}$ programs.

Finally, in Section 6, we compare our language with the formalisms underpinning several existing rule-based systems for data analysis.

This paper summarises the results reported in two conference publications [14, 15], and we refer to them for further details.

2. DATALOG$_{\mathbb{Z}}$

We first recapitulate the well-known definition of Datalog with stratified negation and arithmetic over the set of integers $\mathbb{Z}$, which we call $Datalog_{\mathbb{Z}}$. Our formalism is standard and closely related to constraint logic programming (CLP) over the structure $(\mathbb{Z}, \leq, <, +, -, \times, 0, \pm 1, \pm 2, \dots)$ [7, 8].

We assume countably infinite and mutually disjoint sets of *objects*, *object variables*, *numeric variables*, and *predicates*. Each predicate has a nonnegative integer *arity*, and each position of a predicate is of *object* or *numeric sort*. We call predicates $\leq$ and $<$ with two numeric positions *comparison* predicates, and we call all remaining predicates *standard*. An *object term* is an object or an object variable. A *numeric term* is an integer, a numeric variable, or an expression 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 usual *arithmetic functions*. A *constant* is an object or an integer. A *standard atom* is an expression of the form $A(t_1, \dots, t_v)$, where A is a standard predicate of arity v and each t_i is a term of

the sort of position i in A. A *comparison atom* is an expression of the form $(s_1 \leq s_2)$ or $(s_1 < s_2)$, where s_1 and s_2 are numeric terms. We use $(s_1 \geq s_2)$ for $(s_2 \leq s_1)$, $(s_1 \doteq s_2)$ for $(s_1 \leq s_2) \wedge (s_2 \leq s_1)$, and so on. A *positive literal* is a standard or a comparison atom, a *negative literal* is an expression of the form $\mathsf{not}\,\alpha$ for α a standard atom, and a *literal* is a positive or a negative literal.

A *rule* ρ is of the form $\varphi \rightarrow \alpha$, where the *head* α of ρ is a standard atom and the *body* φ of ρ is a conjunction of literals. We consider only *safe* rules, where each variable occurs in a positive body literal. A *fact* is a rule with the empty body and in which all terms are constants (i.e., it mentions neither variables nor arithmetic functions); we usually omit '$\rightarrow$' in facts. A *dataset* is a finite set of facts.

We use the usual *stratification* condition [8] to ensure that negation is 'well-behaved'. A finite set $\mathcal{P}$ of rules is *stratifiable* if it can be partitioned into disjoint subsets $\mathcal{P}[1], \ldots, \mathcal{P}[h]$ called *strata* such that, for each $i \in [1, h]$, each predicate occurring in $\mathcal{P}[i]$ does not occur in the head of a rule in any $\mathcal{P}[j]$ with $j > i$, and each predicate occurring in a negative body literal of a rule in $\mathcal{P}[i]$ does not also occur in the head of a rule in $\mathcal{P}[i]$. When such a stratification exists, we say that $\mathcal{P}$ *admits* h strata.

A ($Datalog_{\mathbb{Z}}$) *program* $\mathcal{P}$ is a finite stratifiable set of rules. A standard predicate A is *intensional* (*IDB*) in $\mathcal{P}$ if it occurs in $\mathcal{P}$ in the head of a rule that is not a fact; otherwise, A is *extensional* (*EDB*) in $\mathcal{P}$. Program $\mathcal{P}$ is *positive* if it does not use negative literals (so $\mathcal{P}$ admits a single stratum), and it is *semi-positive* if the predicate of each negative literal is EDB in $\mathcal{P}$ (thus, $\mathcal{P}$ admits two strata).

We discuss the semantics of $Datalog_{\mathbb{Z}}$ only informally as it is the same as for usual Datalog with stratified negation [8]. An *interpretation* $\mathcal{I}$ is a (not necessarily finite) set of facts. If all the rules of a program $\mathcal{P}$ are satisfied in $\mathcal{I}$ (under the usual semantics of first-order logic with integer arithmetic, assuming that all variables in rules are universally quantified), then $\mathcal{I}$ is a *model* of $\mathcal{P}$ and we write $\mathcal{I} \models \mathcal{P}$. Since $\mathcal{P}$ is stratified, there exists a unique model $\mathcal{M}(\mathcal{P})$ of $\mathcal{P}$ that is the smallest with respect to set inclusion, which we call the *materialisation* of $\mathcal{P}$. This name is justified by the fact that $\mathcal{M}(\mathcal{P})$ can be constructed by iteratively applying the rules of $\mathcal{P}$ stratum by stratum. Specifically, to apply a rule ρ to an interpretation $\mathcal{I}$, we evaluate the body of ρ as a query over $\mathcal{I}$, and, for each query answer, we instantiate the head of ρ and add the resulting fact to $\mathcal{I}$. Then, we can compute $\mathcal{M}(\mathcal{P})$ bottom-up as follows: after initialising $\mathcal{M}(\mathcal{P})$ by the empty set, we consider the strata of $\mathcal{P}$ one by one so that, for each i in the increasing order, we first apply the rules of the stratum $\mathcal{P}[i]$ to $\mathcal{M}(\mathcal{P})$ as long as possible (i.e., until no new facts can be derived) and then move on to the next stratum $\mathcal{P}[i+1]$. Program $\mathcal{P}$ *entails* a fact γ, written $\mathcal{P} \models \gamma$, if $\gamma \in \mathcal{M}(\mathcal{P})$ holds. Given such $\mathcal{P}$ and γ, checking whether $\mathcal{P} \models \gamma$ holds is a key problem in Datalog and $Datalog_{\mathbb{Z}}$ applications, and it is the main subject of this paper.

If program $\mathcal{P}$ does not use arithmetic functions, then such construction of $\mathcal{M}(\mathcal{P})$ always terminates, and this procedure is used for checking fact entailment in many practical Datalog engines. This, however, no longer holds if $\mathcal{P}$ uses arithmetic.

EXAMPLE 2.1. Let $\mathcal{P}$ be a $Datalog_{\mathbb{Z}}$ program containing a fact $B(0)$ and rule $B(m) \rightarrow B(m+1)$, where predicate B has a single numeric position. Applying $\mathcal{P}$ iteratively derives $B(1), B(2), \ldots$ without stopping. As a result, the materialisation $\mathcal{M}(\mathcal{P})$ contains $B(k)$ for each $k \geq 0$ and is thus infinite. $\lhd$

Despite Example 2.1, the construction of $\mathcal{M}(\mathcal{P})$ is still well defined if we consider the possibly infinite 'limit' of rule application for each $\mathcal{P}[i]$. However, such a 'computation' of $\mathcal{M}(\mathcal{P})$ does not give us an algorithm for checking fact entailment in $Datalog_{\mathbb{Z}}$ with full arithmetic. Moreover, one can exploit the fact that $\mathcal{M}(\mathcal{P})$ can be infinite to show that checking fact entailment is undecidable even for positive programs without multiplication and subtraction that use standard predicates with at most one numeric position [8, 14]. Our goal is thus to identify restrictions that, on the one hand, provide us with languages rich enough to capture interesting data analysis problems and, on the other hand, support decidable or even tractable fact entailment.

3. LIMIT-LINEAR DATALOG$_{\mathbb{Z}}$

In this section, we first introduce *limit* $Datalog_{\mathbb{Z}}$, which can be seen as either a semantic or a syntactic restriction of $Datalog_{\mathbb{Z}}$. To overcome the undecidability of entailment, we then restrict the use of multiplication and thus arrive to *limit-linear* programs. Finally, we present several application examples.

3.1 Limit Programs

As illustrated in Example 2.1, one of the main problems in $Datalog_{\mathbb{Z}}$ is that the materialisation of a program can be infinite. Towards a decidable fragment of $Datalog_{\mathbb{Z}}$, we first introduce *limit* programs. As we shall see, the materialisations of such programs can be represented using finite structures.

DEFINITION 3.1. *A predicate is* object *if it has only object positions, and it is* numeric *if its last*

position is numeric and all its other positions are object; moreover, each numeric predicate is either exact *or* limit*, and each limit predicate is either* min *or* max*. A* limit *(Datalog$_\mathbb{Z}$) program is a program that uses only object and numeric predicates, and where all exact predicates are EDB.*

The notions in Definition 3.1 transfer to atoms and literals in the obvious way; for example, a standard atom is max if its predicate is max.

The intuition behind limit predicates is that they keep only the upper, in case of max, or only the lower, in case of min, bounds on the numeric values for each tuple of object arguments. For example, assume that a program $\mathcal{P}$ consists only of facts $C(a, 5)$ and $C(a, 7)$, where C is a numeric predicate and a is an object. If $\mathcal{P}$ is an ordinary *Datalog*$_\mathbb{Z}$ program, then the materialisation $\mathcal{M}(\mathcal{P})$ coincides with $\mathcal{P}$. If, however, $\mathcal{P}$ is limit and C is max, then the semantics of limit *Datalog*$_\mathbb{Z}$ ensures that every model of $\mathcal{P}$ contains the fact $C(a, k)$ for each $k \leq 7$, and moreover $\mathcal{M}(\mathcal{P})$ consists precisely of these facts. Thus, 7 is the limit value of C on a in $\mathcal{M}(\mathcal{P})$, and we can finitely represent $\mathcal{M}(\mathcal{P})$ as just $C(a, 7)$.

The semantics of a limit program $\mathcal{P}$ is defined model theoretically by considering only *limit-closed* interpretations—that is, interpretations $\mathcal{I}$ that, for each fact $C(\mathbf{a}, \ell) \in \mathcal{I}$ with C a max predicate (in $\mathcal{P}$), contain $C(\mathbf{a}, k)$ for each $k \leq \ell$; and analogously for min predicates. Alternatively, the semantics of limit predicates can be axiomatised in *Datalog*$_\mathbb{Z}$ by extending the program with Rule (3) for each max predicate C and analogously for min predicates.

$$C(\mathbf{x}, m) \wedge (n \leq m) \rightarrow C(\mathbf{x}, n) \tag{3}$$

We introduce a useful syntactic shortcut: for C a limit predicate, $\mathbf{t}$ a tuple of object terms of appropriate size, and s a numeric term, the *least upper bound* (*LUB*) expression $\lceil C(\mathbf{t}, s) \rceil$ abbreviates $C(\mathbf{t}, s) \wedge \mathsf{not}\, C(\mathbf{t}, s + k)$, where $k = 1$ if C is max and $k = -1$ if C is min. Since LUB expressions contain negative literals, $\lceil C(\mathbf{t}, s) \rceil$ can be used in a stratum $\mathcal{P}[i]$ of a limit program $\mathcal{P}$ only if C does not occur in a rule head in $\mathcal{P}[j]$ for each $j \geq i$.

The following example illustrates the intuitions of the definitions we presented thus far.

Example 3.2. Let $\mathcal{P}$ be the program containing Rules (1) and (2) from Section 1, and the facts that describe a directed graph with edge lengths using predicate E. When computing the lengths of the shortest paths from a_s, we need not remember the length of each path from a_s: it suffices to keep just the lengths of the shortest paths found so far. Thus, we can make *dst* a min predicate, which is tantamount to extending $\mathcal{P}$ with the following rule.

$$dst(x, m) \wedge (m \leq n) \rightarrow dst(x, n)$$

As a consequence of this change, a fact $dst(a, \ell)$ follows from $\mathcal{P}$ if and only if the distance from the source node a_s to a is *at most* ℓ; hence, each $dst(a, k)$ with $k \geq \ell$ then follows as well. Note that only *dst* is a limit predicate: predicate E is EDB and so it can be exact, which is in fact necessary to correctly encode the graph structure. Finally, using an LUB expression we can query the exact length of a shortest path: $\mathcal{P}$ entails $\lceil dst(a, \ell) \rceil$ if and only if ℓ is the exact length from a_s to a. ◁

We now discuss the technical challenges of dealing with limit predicates. First, note that any limit-closed interpretation containing a fact over a limit predicate is infinite. In particular, the materialisation of the program from Example 2.1 does not change even if we make B a min predicate. A key insight is that the interpretation of a limit predicate is 'contiguous' for each tuple of object arguments; hence, instead of keeping all of these facts, we can remember only the limit values. Moreover, the limit value may not exist; for example, if we make B a max predicate in Example 2.1, then $\mathcal{M}(\mathcal{P})$ contains $B(k)$ for each integer k. However, we can represent such cases using a special symbol ∞. This motivates the following definition.

Definition 3.3. *A* pseudofact *is either a fact or an expression of the form $C(\mathbf{a}, \infty)$ for C a limit predicate and $\mathbf{a}$ a tuple of objects. A* pseudointerpretation *is a set $\mathcal{J}$ of pseudofacts such that $\ell_1 = \ell_2$ for all limit pseudofacts $C(\mathbf{a}, \ell_1)$ and $C(\mathbf{a}, \ell_2)$ in $\mathcal{J}$.*

We stress an important point of Definition 3.3. Intuitively, pseudofacts represent limit values, so two different pseudofacts for the same predicate and object arguments should not appear in any pseudointerpretation together; for example, a pseudointerpretation for the program in Example 3.2 can contain either $dst(a, 5)$ or $dst(a, 7)$, but never both. This, however, leads us to an important observation: a pseudointerpretation is finite whenever the numbers of predicates and object constants are finite. This property of pseudointerpretations is essential for our decidability results in Section 4.

Moreover, it is easy to see that limit-closed interpretations and pseudointerpretations naturally correspond to each other. For example, to convert a pseudointerpretation $\mathcal{J}$ to a limit-closed interpretation $\mathcal{I}$, we replace each max (pseudo)fact $C(\mathbf{a}, \ell)$ in $\mathcal{J}$ with $\ell \in \mathbb{Z}$ by all facts $C(\mathbf{a}, k)$ with $k \leq \ell$, each such min fact by all $C(\mathbf{a}, k)$ with $k \geq \ell$, and each limit pseudofact $C(\mathbf{a}, \infty)$ by all $C(\mathbf{a}, k)$ with $k \in \mathbb{Z}$.

Conversion in the other direction can be done in a similar way. This allows us to transfer all definitions and notations for limit-closed interpretations to pseudointerpretations; for example, a pseudointerpretation $\mathcal{J}$ *entails* a fact γ if the corresponding limit-closed interpretation $\mathcal{I}$ entails γ, $\mathcal{J}$ is a *pseudomodel* of a limit program $\mathcal{P}$ if $\mathcal{I} \models \mathcal{P}$, and the *pseudomaterialisation* $\mathcal{N}(\mathcal{P})$ is the pseudointerpretation corresponding to the materialisation $\mathcal{M}(\mathcal{P})$.

Finally, we observe that each limit program can be easily rewritten into an equivalent *homogeneous* program that uses only max (or only min) predicates. This can be done by replacing all min predicates with fresh max predicates of the same arities and negating the corresponding numeric arguments in atoms with the replaced predicates.

3.2 Limit-Linear Programs

The ability to finitely represent materialisations does not, however, ensure decidability.

THEOREM 3.4. *The fact entailment problem for positive limit programs is undecidable.*

Theorem 3.4 is due to the fact that limit programs allow multiplication of numeric variables, which we use to reduce the well-known undecidable problem of solving Diophantine equations (i.e., finding integer roots of polynomials) to fact entailment. This motivates the following *linearity* restriction.

DEFINITION 3.5. *A numeric variable m is* guarded *in a rule if it occurs in the rule body in either a function-free positive exact literal or an LUB expression of the form $\lceil C(\mathbf{t}, m)\rceil$. A rule or program is* limit-linear *(*LL-*) if, in each multiplication, at most one argument mentions an unguarded variable.*

Intuitively, a guarded variable m in a rule of an LL-program $\mathcal{P}$ can be matched only to finitely many integers during the evaluation of $\mathcal{P}$. To see this, first note that all exact predicates are EDB in $\mathcal{P}$; thus, if m is guarded because it occurs in a function-free positive literal over an exact predicate, then m can be matched only to facts explicitly mentioned in $\mathcal{P}$. Second, if m is guarded because it occurs in an LUB expression $\lceil C(\mathbf{t}, m)\rceil$, then variable m can be matched to the limit value of C for each valuation of $\mathbf{t}$; moreover, since $\lceil C(\mathbf{t}, m)\rceil$ abbreviates a conjunction containing $\mathsf{not}\, C(\mathbf{t}, m+k)$ for $k = \pm 1$, the limit values for C are fully determined by the strata of $\mathcal{P}$ preceding the stratum of the rule. Note that atoms with other numeric terms involving m, such as $B(\mathbf{t}, m+1)$, do not make m guarded.

To understand how guarded variables are used to ensure decidability, consider an LL-rule ρ containing a numeric term $m \times n$ with numeric variables m and n. Since ρ is limit-linear, at least one of m and n must be guarded. If m is guarded, then, by the previous paragraph, m can be matched to finitely many integers $k_1, \ldots, k_v$ and hence we can replace ρ with its v instances where the term $m \times n$ is replaced by $k_i \times n$. We have thus reduced multiplication of variables $m \times n$ to *linear* multiplication $k_i \times n$, which allows us to obtain decidability in Section 4 using methods from Presburger arithmetic.

To simplify the discussion in the rest of this paper, we assume that each LL-program is normalised so that each exact atom is function-free. This can be achieved by replacing each positive exact body atom $B(\mathbf{a}, s)$ with s containing functions by the conjunction $B(\mathbf{a}, m) \wedge (m \doteq s)$ with m a fresh numeric variable. Moreover, we note that all exact atoms in rule heads are function-free: the predicates in all these atoms are EDB, and so all such rules are facts.

3.3 Application Examples

Despite the restrictions of Definitions 3.1 and 3.5, LL-programs can still naturally capture many interesting data analysis tasks. We next present five such examples motivated by practical applications.

EXAMPLE 3.6 (DIFFUSION IN NETWORKS). Consider a social network of agents connected by the 'follows' relation. Agent a_s introduces (*tweets*) a message, and each agent a retweets the message if at least k_a agents that a follows tweet this message, where k_a is a positive threshold associated with a. We can determine which agents tweet the message eventually using limit-linear $Datalog_{\mathbb{Z}}$ as follows. We encode the network structure as a dataset $\mathcal{D}_{tw}$ consisting of the object fact $tw(a_s)$ saying that a_s introduces a message, object facts $fol(a, a')$ saying that a follows a', and exact facts $thshld(a, k_a)$ saying that the threshold of a is k_a. We also assume that $\mathcal{D}_{tw}$ is *ordered*—that is, it contains facts $fst(a_1)$, $nxt(a_1, a_2)$, …, $nxt(a_{c-1}, a_c)$, $lst(a_c)$ for some enumeration $a_1, \ldots, a_c$ of all objects (i.e., agents) in $\mathcal{D}_{tw}$. The LL-program $\mathcal{P}_{tw}$, consisting of Rules (4)–(8), encodes message propagation. Here, ac is an 'accumulating' max predicate such that $ac(a, a', m)$ is *true* if there are at least m agents tweeting the message among the agents that a follows and that (inclusively) precede a' in the dataset order.

$$fol(x, y') \wedge fst(y) \rightarrow ac(x, y, 0) \tag{4}$$

$$tw(y) \wedge fol(x, y) \wedge fst(y) \rightarrow ac(x, y, 1) \tag{5}$$

$$ac(x, y', m) \wedge nxt(y', y) \rightarrow ac(x, y, m) \tag{6}$$

$$tw(y) \wedge fol(x, y) \wedge ac(x, y', m) \wedge nxt(y', y) \rightarrow ac(x, y, m+1) \tag{7}$$

$$thshld(x, m) \wedge ac(x, y, m) \rightarrow tw(x) \tag{8}$$

Then, $\mathcal{P}_{tw} \cup \mathcal{D}_{tw} \models tw(a)$ if and only if an agent a tweets the message according to $\mathcal{D}_{tw}$. $\triangleleft$

Example 3.7 (Bill of materials).
Let $\mathcal{D}_{bm}$ be a dataset describing parts needed to manufacture an end product. Specifically, for each part a and each subpart a' of a, $\mathcal{D}_{bm}$ contains object facts $pt(a)$ and $pt(a')$, and an exact fact $dsp(a, a', k)$ indicating that a needs k copies of a'; also, let $\mathcal{D}_{bm}$ be ordered as in Example 3.6. The graph formed by predicate *dsp* is acyclic and has positive edge weights. Rules (9)–(14) form the LL-program $\mathcal{P}_{bm}$. They compute, using max predicates *ac* and *sp*, how many copies of each subpart are used for each part in total. Intuitively, $ac(a, a', b, k)$ is *true* if part a contains at least k copies of subpart a' in all direct subparts of a that precede part b in the order.

$$pt(x) \to sp(x, x, 1) \tag{9}$$

$$pt(x) \wedge pt(y) \wedge fst(z) \to ac(x, y, z, 0) \tag{10}$$

$$dsp(x, z, n_1) \wedge sp(z, y, n_2) \wedge fst(z) \to ac(x, y, z, n_1 \times n_2) \tag{11}$$

$$ac(x, y, z', m) \wedge nxt(z', z) \to ac(x, y, z, m) \tag{12}$$

$$dsp(x, z, n_1) \wedge sp(z, y, n_2) \wedge ac(x, y, z', m) \wedge nxt(z', z) \to ac(x, y, z, m + n_1 \times n_2) \tag{13}$$

$$ac(x, y, z, m) \to sp(x, y, m) \tag{14}$$

Then, $\mathcal{P}_{bm} \cup \mathcal{D}_{bm} \models sp(a, a', k)$ if and only if a contains at least k copies of a'. Program $\mathcal{P}_{bm}$ is limit-linear since n_1 occurs in positive exact literals over *dsp* and is thus guarded in (11) and (13). $\triangleleft$

Example 3.8 (Counting paths).
Limit-linear $Datalog_{\mathbb{Z}}$ can also count the paths between pairs of nodes in a directed acyclic graph. We encode such a graph in the obvious way as a dataset $\mathcal{D}_{cp}$ that uses a unary object predicate *nd* for nodes and a binary object predicate E for edges; moreover, $\mathcal{D}_{cp}$ is ordered as before. The LL-program $\mathcal{P}_{cp}$, consisting of Rules (15)–(20) with max predicates *pn* and *ac*, counts the paths. Intuitively, $ac(a, a', b, k)$ is *true* if the sum of the numbers of paths from each node b' preceding node b (according to the dataset order) to node a' for which there exists an edge from node a to b' is at least k.

$$nd(x) \to pn(x, x, 1) \tag{15}$$

$$nd(x) \wedge nd(y) \wedge fst(z) \to ac(x, y, z, 0) \tag{16}$$

$$E(x, z) \wedge pn(z, y, n) \wedge fst(z) \to ac(x, y, z, n) \tag{17}$$

$$ac(x, y, z', m) \wedge nxt(z', z) \to ac(x, y, z, m) \tag{18}$$

$$E(x, z) \wedge pn(z, y, n) \wedge ac(x, y, z', m) \wedge nxt(z', z) \to ac(x, y, z, m + n) \tag{19}$$

$$ac(x, y, z, m) \to pn(x, y, m) \tag{20}$$

Then, $\mathcal{P}_{cp} \cup \mathcal{D}_{cp} \models pn(a, a', k)$ if and only if there are at least k paths from node a to node a'. $\triangleleft$

All examples provided thus far use positive LL-programs. In contrast, the following two examples demonstrate the use of stratified negation.

Example 3.9 (Shortest paths).
We modify the program from Section 1 to compute not just the shortest distance, but also the actual paths from a given source node a_s to a given target node a_t in a directed graph with weighted edges. We assume that a dataset $\mathcal{D}_{csp}$ encodes a directed graph with positive edge weights using a ternary exact predicate E as before, and that it identifies the source and target nodes using object facts $src(a_s)$ and $tgt(a_t)$, respectively. The LL-program $\mathcal{P}_{csp}$, consisting of Rule (2) and Rules (21)–(23), computes a directed acyclic graph G with source a_s and target a_t, encoded using a binary object predicate *spE*, such that every maximal path in G is a shortest path from a_s to a_t in the original graph.

$$src(x) \to dst(x, 0) \tag{21}$$

$$\lceil dst(x, m) \rceil \wedge E(x, y, n) \wedge \lceil dst(y, m + n) \rceil \wedge tgt(y) \to spE(x, y) \tag{22}$$

$$\lceil dst(x, m) \rceil \wedge E(x, y, n) \wedge \lceil dst(y, m + n) \rceil \wedge spE(y, z) \to spE(x, y) \tag{23}$$

The first stratum consists of Rules (2) and (21), and computes the length of a shortest path from a_s to each other node using the min predicate *dst*; in particular, we have that $\mathcal{P}_{csp} \cup \mathcal{D}_{csp} \models \lceil dst(a, k) \rceil$ if and only if k is the length of a shortest path from a_s to a. Then, the second stratum, consisting of Rules (22) and (23), computes predicate *spE* such that $\mathcal{P}_{csp} \cup \mathcal{D}_{csp} \models spE(a, a')$ if the edge (a, a') is a part of a shortest path from a_s to a_t. $\triangleleft$

Example 3.10 (Closeness centrality).
The closeness centrality of a node in a strongly connected directed graph G with weighted edges is a measure of how central the node is in the graph [23]. Variants of this measure are useful, for example, for the analysis of market potential. The *closeness centrality* of a node a is $1 \,/\, \sum_{a' \text{ node in } G} \Omega(a, a')$, where $\Omega(a, a')$ is the length of a shortest path from a to a'; the sum in the denominator is often called the *farness centrality* of a. We next present an LL-program $\mathcal{P}_{cc}$ that identifies a node of maximal closeness centrality in a strongly connected directed graph with weighted edges. We encode such a graph as an ordered dataset $\mathcal{D}_{cc}$ using a unary object predicate *nd* and a ternary exact predicate E. Program

$\mathcal{P}_{cc}$ consists of Rules (24)–(32), where *dst*, *fns* and *fns′* are min, and *cntr* and *cntr′* are object.

$$nd(x) \to dst(x, x, 0) \tag{24}$$
$$dst(x, y, m) \wedge E(y, z, n) \to dst(x, z, m{+}n) \tag{25}$$
$$nd(x) \wedge fst(y) \wedge dst(x, y, n) \to fns'(x, y, n) \tag{26}$$
$$fns'(x, y, m) \wedge nxt(y, z) \wedge dst(x, z, n) \to fns'(x, z, m{+}n) \tag{27}$$
$$fns'(x, y, n) \wedge lst(y) \to fns(x, n) \tag{28}$$
$$fst(x) \to cntr'(x, x) \tag{29}$$
$$cntr'(x, z) \wedge nxt(x, y) \wedge \lceil fns(z, m) \rceil \wedge \lceil fns(y, n) \rceil \wedge (n < m) \to cntr'(y, y) \tag{30}$$
$$cntr'(x, z) \wedge nxt(x, y) \wedge \lceil fns(z, m) \rceil \wedge \lceil fns(y, n) \rceil \wedge (m \leq n) \to cntr'(y, z) \tag{31}$$
$$cntr'(x, z) \wedge lst(x) \to cntr(z) \tag{32}$$

The first stratum consists of Rules (24)–(28); Rules (24) and (25) compute the distance between any two nodes, and Rules (26)–(28) then compute the farness centrality of each node based on the aforementioned distances. In the second stratum, (29)–(32), $\mathcal{P}_{cc}$ uses negation (hidden inside the LUB expressions) to compute a node of minimal farness centrality (and hence of maximal closeness centrality), which is recorded using the *cntr* predicate. ◁

4. COMPLEXITY OF LL-PROGRAMS

We now discuss the complexity of fact entailment in limit-linear $Datalog_{\mathbb{Z}}$. We consider *combined complexity*, where the input consists of an LL-program $\mathcal{P}$ and a fact γ, and *data complexity*, where we assume that $\mathcal{P} = \mathcal{P}' \cup \mathcal{D}$ for a dataset $\mathcal{D}$ and a fixed LL-program $\mathcal{P}'$ (i.e., only $\mathcal{D}$ and γ are given as input). We assume binary coding of integers and write $\|\mathcal{P}\|$ for the size of the representation of a program $\mathcal{P}$; however, our results also hold for unary coding. We show that, for the full language, the problem is complete for $\Delta_2^{\mathsf{EXP}} = \mathsf{EXP}^{\mathsf{NP}}$ in combined and for $\Delta_2^{\mathsf{P}} = \mathsf{P}^{\mathsf{NP}}$ in data complexity, while, for the positive and semi-positive fragments, it is complete for coNEXP and for coNP, respectively. Thus, (semi-)positive LL-programs have the same complexity as answer-set programming [24], and are one level above the usual (semi-)positive Datalog, which is EXP- and P-complete [8]. Moreover, in terms of complexity, LL-programs are between the usual and disjunctive answer-set programming, where the latter is complete for Π_2^{EXP} and Π_2^{P} [10].

4.1 Positive Fragment

We first consider the problem of checking entailment $\mathcal{P} \models \gamma$ of a fact γ by a positive LL-program $\mathcal{P}$. Reasoning algorithms for usual Datalog often start with computing the program *grounding*—that is, replacing all variables with constants in the program in all possible ways. This variable elimination simplifies rule application, but with a cost of an exponential blow-up (in the size of the program). Our algorithms use a similar preprocessing step. However, grounding in a usual way would require replacing each numeric variable in an LL-program with every integer, which would result in an infinite grounding. To avoid working with infinite programs, we need to adapt the notion of grounding.

Definition 4.1. *An LL-rule or LL-program is* object-and-guarded-ground *(*OG-ground*) if it has neither object nor guarded numeric variables. The* canonical OG-grounding *of an LL-program $\mathcal{P}$ is the OG-ground program $\mathcal{G}(\mathcal{P})$ that contains the OG-ground instance $\rho\sigma$ for each rule $\rho \in \mathcal{P}$ and each substitution σ mapping all object and guarded numeric variables of ρ to constants in $\mathcal{P}$.*

To produce $\mathcal{G}(\mathcal{P})$ for a positive LL-program $\mathcal{P}$, we replace, in all possible ways, all object variables in $\mathcal{P}$ with objects in $\mathcal{P}$ and all guarded numeric variables with integers in $\mathcal{P}$. The discussion in Section 3.2 explains why considering only the integers from $\mathcal{P}$ suffices. It is easy to see that $\mathcal{P}$ and $\mathcal{G}(\mathcal{P})$ are equivalent in the sense that $\mathcal{M}(\mathcal{P}) = \mathcal{M}(\mathcal{G}(\mathcal{P}))$, so $\mathcal{P} \models \gamma$ if and only if $\mathcal{G}(\mathcal{P}) \models \gamma$ for each fact γ.

Now we show how to apply a positive OG-ground rule ρ to a pseudointerpretation $\mathcal{J}$. A minor problem is that, if we derive a fact $C(\mathbf{a}, \ell)$ with C max and we already have a fact $C(\mathbf{a}, k)$ with $k < \ell$, then we must remove $C(\mathbf{a}, k)$, and similarly if C is min. More importantly, identifying the substitutions that match the body of ρ to $\mathcal{J}$ is considerably more complex than for ordinary interpretations. For example, the body of the rule $C_1(m) \wedge C_2(m) \to C(m)$ where C_1 and C_2 are max predicates does not match to the pseudointerpretation $\{C_1(7), C_2(5)\}$ directly, but the rule is applicable and it derives $C(5)$. We address this difficulty by reducing the problem of a positive OG-ground rule application to *integer linear optimisation*. Specifically, to match ρ to $\mathcal{J}$, we can transform ρ into a system of integer linear inequalities $\psi(\rho, \mathcal{J})$ whose solutions encode exactly the substitutions obtained by matching of the body of ρ to (the limit-closed interpretation corresponding to) $\mathcal{J}$. Thus, to compute the consequences of ρ with a limit atom in the head, we just need the solution that optimises the numeric term in this atom.

Unfortunately, as shown in Example 2.1, iterative rule application may not terminate even for OG-ground programs. So, while one can formally

describe a process that constructs the (finite) pseudomaterialisation by iteratively applying rules, this process may be infinite and thus does not provide us with a decision procedure for fact entailment.

We next discuss a key insight about positive LL-programs, which we use to bound the magnitudes of integers in pseudomaterialisations. We exploit a connection with *Presburger arithmetic*—a theory of first-order formulas with numeric variables (i.e., all variables range over integers) where all atoms are comparisons of the form $(s_1 \leq s_2)$ or $(s_1 < s_2)$ (i.e., as in $Datalog_{\mathbb{Z}}$) with multiplication-free numeric terms s_1 and s_2. Due to the lack of multiplication, reasoning in this theory is decidable.

Now, to check entailment $\mathcal{P} \models \gamma$ for a positive LL-program $\mathcal{P}$ and a fact γ, we encode $\mathcal{P}$ as a Presburger formula $\xi_{\mathcal{P}}$ so that each pseudomodel of $\mathcal{P}$ corresponds to a valuation of the free variables of $\xi_{\mathcal{P}}$ that makes $\xi_{\mathcal{P}}$ true. We analogously encode γ as a formula ξ_γ, and we thus reduce $\mathcal{P} \models \gamma$ to checking whether the Presburger sentence $\forall \mathbf{v}. (\xi_{\mathcal{P}} \rightarrow \xi_\gamma)$, where $\mathbf{v}$ are the free variables of $\xi_{\mathcal{P}}$ and ξ_γ, is true. To illustrate a simplified version of our encoding, consider an LL-program $\mathcal{P}$ consisting of a fact $C(2)$ and a rule $C(m) \rightarrow D(m+2)$, and a fact $\gamma = D(3)$, where both C and D are max. We encode the values of C and D in a pseudomodel of $\mathcal{P}$ using variables v_C and v_D, respectively. Fact $C(2)$ says 'the value of C in each pseudomodel is at least 2', so we encode it as $\xi_1 = (2 \leq v_C)$. We also encode the rule as $\xi_2 = \forall m. (m \leq v_C) \rightarrow (m + 2 \leq v_D)$, and γ as $\xi_\gamma = (3 \leq v_D)$. Clearly, $\mathcal{P} \models \gamma$ if and only if $\forall v_C \forall v_D. (\xi_{\mathcal{P}} \rightarrow \xi_\gamma)$ is true for $\xi_{\mathcal{P}} = \xi_1 \wedge \xi_2$. Following this idea, our encoding uses additional variables to represent undefined and unbounded values.

Thus, entailment $\mathcal{P} \models \gamma$ for a positive LL-program $\mathcal{P}$ and fact γ is decidable since Presburger arithmetic is decidable; however, the complexity bounds derived from the standard reasoning algorithms for Presburger arithmetic are not optimal. Instead, by analysing the structure of our encoding and applying the results by Chistikov and Haase [5], we show that $\forall \mathbf{v}. (\xi_{\mathcal{P}} \rightarrow \xi_\gamma)$ is true if and only if it is true when the sentence is evaluated over integers with magnitudes at most double exponential in $\|\mathcal{P}\| + \|\gamma\|$, and at most exponential in $\|\mathcal{D}\| + \|\gamma\|$, where $\mathcal{D}$ is the dataset part of $\mathcal{P}$.[1] So, each integer can be written in binary using number of bits exponential in $\|\mathcal{P}\| + \|\gamma\|$ and polynomial in $\|\mathcal{D}\| + \|\gamma\|$.

Since the valuations of the free variables of $\xi_{\mathcal{P}}$ encode the pseudomodels of $\mathcal{P}$, nonentailment $\mathcal{P} \not\models \gamma$ is witnessed by a pseudomodel $\mathcal{J}$ of $\mathcal{P}$ where the integers are bounded in the same way (note that ∞ is not an integer so $\mathcal{J}$ can contain ∞). Thus, we can decide $\mathcal{P} \not\models \gamma$ by first guessing a pseudointerpretation $\mathcal{J}$ over the signature of $\mathcal{P}$ with integers bounded as explained, and then checking that $\mathcal{J} \models \mathcal{G}(\mathcal{P})$ and $\mathcal{J} \not\models \gamma$. Overall, $\|\mathcal{J}\|$ is at most exponential in $\|\mathcal{P}\| + \|\gamma\|$ and at most polynomial in $\|\mathcal{D}\| + \|\gamma\|$. Moreover, to check $\mathcal{J} \models \mathcal{G}(\mathcal{P})$, we apply the rules of $\mathcal{G}(\mathcal{P})$ to $\mathcal{J}$ and verify that no new fact is derived, which requires solving integer optimisation problems. Thus, our algorithm works in NEXP in combined and in NP in data complexity.

For the matching lower data complexity bound, we reduce the complement of the canonical NP-complete problem SAT. Given an arbitrary propositional formula Φ with h variables, we (arbitrarily) order all variables of Φ so we can view each variable assignment σ as an integer $\ell(\sigma)$ between 0 and $2^h - 1$. To decide the satisfiability of Φ, we then can enumerate all assignments σ in the increasing order of $\ell(\sigma)$ until we either find a σ satisfying Φ, or we find that σ_{max} with $\ell(\sigma_{max}) = 2^h - 1$ does not satisfy Φ. If Φ is encoded in a dataset, this enumeration can be simulated by a fixed positive LL-program that stores $\ell(\sigma)$ in the numeric argument of a max predicate, starting with 0 and incrementing if and only if the current σ does not satisfy Φ. Hence, Φ is unsatisfiable if and only if our encoding entails a fact with a numeric argument 2^h.

As required for data complexity, the program in this reduction does not depend on input Φ. In contrast, this is not necessary for combined complexity, and the same ideas can be applied to reduce the succinct version of SAT—a canonical NEXP-complete problem [21]. We arrive to the following result.

THEOREM 4.2. *The fact entailment problem for positive LL-programs is* coNEXP*-complete in combined and* coNP*-complete in data complexity.*

4.2 Semi-Positive and Full Fragments

We first consider semi-positive programs, where negation can be used only over EDB predicates. We reduce our problem to the positive case by computing the canonical OG-grounding of the given program and 'evaluating' all negative literals. This idea is captured by the following definition.

DEFINITION 4.3. *The* reduct $\mathcal{R}(\mathcal{P})$ *of a semi-positive LL-program $\mathcal{P}$ is the positive OG-program obtained from $\mathcal{G}(\mathcal{P})$ by applying the following to each rule $\rho \in \mathcal{G}(\mathcal{P})$ and each negative literal* not α *in ρ.*

- *Let α be an object atom. If $\alpha \notin \mathcal{G}(\mathcal{P})$, then delete* not α *from ρ, and otherwise delete ρ.*
- *Let $\alpha = B(\mathbf{a}, s)$ be an exact atom and consider all integers $k_1 < \cdots < k_h$ such that $B(\mathbf{a}, k_i) \in \mathcal{G}(\mathcal{P})$*

[1] We thank Christoph Haase for providing the proof of a key lemma for this statement.

holds for each $i \in [1, h]$. *If* $h = 0$, *then delete* $\mathsf{not}\,\alpha$ *from* ρ; *otherwise, replace* ρ *by* $h + 1$ *rules obtained by replacing* $\mathsf{not}\,\alpha$ *in* ρ *with* $(s < k_1)$, $(k_{i-1} < s < k_i)$ *for* $i \in [2, h]$, *and* $(k_h < s)$.

– *Let* $\alpha = C(\mathbf{a}, s)$ *be a limit atom and consider the set* $S = \{\ell \in \mathbb{Z} \mid C(\mathbf{a}, \ell) \in \mathcal{G}(\mathcal{P})\}$ *of integers. If* $S = \emptyset$, *then delete* $\mathsf{not}\,\alpha$ *from* ρ; *otherwise, replace* $\mathsf{not}\,\alpha$ *in* ρ *with atom* $(\max S < s)$ *if* C *is* max, *and with atom* $(\min S > s)$ *if* C *is* min.

By construction, $\mathcal{M}(\mathcal{P}) = \mathcal{M}(\mathcal{R}(\mathcal{P}))$ for each semi-positive LL-program $\mathcal{P}$, so $\mathcal{P} \models \gamma$ is equivalent to $\mathcal{R}(\mathcal{P}) \models \gamma$ for each fact γ. Moreover, $\mathcal{R}(\mathcal{P})$ is positive and OG-ground, and can be computed with the same bounds as $\mathcal{G}(\mathcal{P})$. Thus, the upper bounds of Theorem 4.2 transfer to semi-positive programs.

THEOREM 4.4. *The fact entailment problem for semi-positive LL-programs is in* coNEXP *in combined and in* coNP *in data complexity.*

To handle arbitrary LL-programs, we first show that the integer bound from Section 4.1 applies not only to some pseudomodel, but also to the pseudo-materialisation of each positive LL-program. Then, given a fact γ and an arbitrary LL-program $\mathcal{P}$ with strata $\mathcal{P}[1], \dots, \mathcal{P}[h]$, we decide $\mathcal{P} \models \gamma$ as follows. For each $i \in [1, h]$, first, we let $\mathcal{P}'_i = \mathcal{R}(\mathcal{P}[i] \cup \mathcal{J}_{i-1})$ (assuming $\mathcal{J}_0 = \emptyset$); then, we add to the pseudointerpretation $\mathcal{J}_i$ each max pseudofact $C(\mathbf{a}, \ell)$ such that $\mathcal{P}'_i \models C(\mathbf{a}, \ell)$ and ℓ is either ∞ or an integer bounded as above satisfying $\mathcal{P}'_i \not\models C(\mathbf{a}, \ell + 1)$; finally, we add to $\mathcal{J}_i$ pseudofacts of other types analogously. The pseudofacts in $\mathcal{P}[i] \cup \mathcal{J}_{i-1}$ can contain symbol ∞, but Definition 4.3 generalises to such 'programs' without change. Clearly, $\mathcal{J}_h = \mathcal{N}(\mathcal{P})$, and so $\mathcal{P} \models \gamma$ if and only if $\mathcal{J}_h \models \gamma$. A naïve complexity bound of this algorithm is nonelementary, but a fine-grained analysis (in particular, bounding $\|\mathcal{J}_i\|$) gives the upper bounds of Theorem 4.5.

For the lower bounds, we generalise the ideas of the positive case. In particular, for data complexity, we reduce the canonical Δ_2^{P}-complete problem ODD-MAX-SAT, where the question is if the maximum value of $\ell(\sigma)$ over all assignments σ satisfying a propositional formula Φ is odd. For the combined complexity, we use a similar reduction of the Δ_2^{EXP}-complete succinct version of ODD-MAX-SAT.

THEOREM 4.5. *The fact entailment problem for LL-programs is* Δ_2^{EXP}*-complete in combined and* Δ_2^{P}*-complete in data complexity.*

5. TRACTABLE FRAGMENTS

Tractability of reasoning in data complexity is important for problems involving large datasets. We now present a *stability* condition on LL-programs that brings the complexity of reasoning down to EXP in combined and to P in data complexity, thus matching the bounds for usual Datalog. We then present a syntactic *type-consistency* condition that ensures stability and is simple to check.

5.1 Stable LL-Programs

The fact entailment algorithm for usual Datalog computes the materialisation iteratively, which can be done in polynomial time in the size of data. We now present a further restriction on LL-programs that makes such iterative computation viable for LL-programs. The key difficulty in doing so is to detect when a numeric argument *diverge*—that is, when it increases or decreases indefinitely. Hence, to ensure tractability, we must be able to detect divergence after polynomially many rule applications. Example 5.1 illustrates the problem of divergence.

EXAMPLE 5.1. Consider an LL-program $\mathcal{P}_{st}$ with the following rules with max predicates C_1 and C_2.

$$C_1(0) \quad C_1(m) \to C_2(m) \quad C_2(m) \to C_1(m+1)$$

Both C_1 and C_2 diverge when computing pseudo-materialisation $\mathcal{N}(\mathcal{P}_{st})$ due to a cyclic dependency between C_1 and C_2. The existence of such a dependency, however, may not lead to divergence. Let an LL-program $\mathcal{P}'_{st}$ be obtained from $\mathcal{P}_{st}$ by adding a fact $C(5)$, for C max, and replacing the second rule by the following rule.

$$C_1(m) \wedge C(m) \to C_2(m)$$

While C_1 and C_2 still depend on each other, the increase in C_1 and C_2 is bounded by an independent value of C, so neither C_1 nor C_2 diverges. ◁

To capture these ideas formally, we first extend $\mathbb{Z} \cup \{\infty\}$ with a new symbol $\bot$, which indicates that a fact does not hold for any integer. We also define $\bot < k < \infty$ for each $k \in \mathbb{Z}$; $\bot + \ell = \bot$ and $\infty + \ell = \infty$ for each $\ell \in \mathbb{Z} \cup \{\infty\}$; and $\bot + \infty = \bot$.

Next we formalise the notion of dependency: a numeric variable m *depends* on a numeric variable n in an OG-ground rule ρ if $m = n$ or m occurs in an atom in ρ with a variable that depends on n. A numeric term s_2 *depends* on a numeric term s_1 if s_2 mentions a variable depending on a variable in s_1.

We next introduce a key notion of a value propagation graph. Our definition is based on the system of linear inequalitites $\psi(\rho, \mathcal{J})$ from Section 4.1 whose solutions encode matches of the body of an OG-ground rule ρ to a pseudointerpretation $\mathcal{J}$. So, if the head of ρ is a limit atom $C(\mathbf{a}, s)$, then $\psi(\rho, \mathcal{J})$

corresponds to an integer linear optimisation problem $\psi^*(\rho, \mathcal{J})$ that optimises (i.e., maximises or minimises, depending on the type of C) the value of s under $\psi(\rho, \mathcal{J})$. If $\psi(\rho, \mathcal{J})$ is satisfiable (i.e., ρ is applicable to $\mathcal{J}$), then $\psi^*(\rho, \mathcal{J})$ can either be bounded and have an optimal integer value, or unbounded.

Definition 5.2. *Given an OG-ground program $\mathcal{P}$ and a pseudointerpretation $\mathcal{J}$, the* value propagation graph *of $\mathcal{P}$ over $\mathcal{J}$ is the weighted directed graph (V, E, Ω) with the following components.*

- *The set of nodes V contains a node $\langle C\mathbf{a}\rangle$ for each limit atom $C(\mathbf{a}, s)$ in a rule head in $\mathcal{P}$.*
- *The set of edges E contains $(\langle C_1\mathbf{a}_1\rangle, \langle C_2\mathbf{a}_2\rangle)$ for each rule ρ in $\mathcal{P}$ with satisfiable $\psi(\rho, \mathcal{J})$ that* produces *the edge—that is, has $C_1(\mathbf{a}_1, s_1)$ in the body and $C_2(\mathbf{a}_2, s_2)$ in the head where s_2 depends on s_1.*
- *The weight $\Omega(e)$ of each edge $e = (\langle C_1\mathbf{a}_1\rangle, \langle C_2\mathbf{a}_2\rangle)$ in E is an element of $\mathbb{Z} \cup \{\bot, \infty\}$ defined as*

$$\Omega(e) = \max\{\Omega_\rho(e) \mid \rho \in \mathcal{P} \text{ produces } e\},$$

where $\Omega_\rho(e)$ is defined as follows, for $\ell \in \mathbb{Z} \cup \{\infty\}$ with $C_1(\mathbf{a}_1, \ell) \in \mathcal{J}$ (which exists by applicability):
  - *$\Omega_\rho(e) = \infty$ if $\psi^*(\rho, \mathcal{J})$ is unbounded,*
  - *$\Omega_\rho(e) = \bot$ if $\psi^*(\rho, \mathcal{J})$ is bounded and $\ell = \infty$,*
  - *$\Omega_\rho(e) = (-1)^{d_2} \cdot k - (-1)^{d_1} \cdot \ell$ if $\psi^*(\rho, \mathcal{J})$ has optimal value k and $\ell \in \mathbb{Z}$ where, for $i \in \{1, 2\}$, d_i is 0 if C_i is* max *and 1 if C_i is* min.

The weight $\Omega(\Pi)$ *of a path Π in a value propagation graph is the sum of the edge weights along Π; Π has* positive weight *if $\Omega(\Pi)$ is a positive integer or ∞.*

Intuitively, graph (V, E, Ω) of a OG-ground program $\mathcal{P}$ over a pseudointerpretation $\mathcal{J}$ describes how, for each pseudofact $C_1(\mathbf{a}_1, \ell)$ in $\mathcal{J}$, applying $\mathcal{P}$ propagates ℓ to other pseudofacts. For example, every edge $e = (\langle C_1\mathbf{a}_1\rangle, \langle C_2\mathbf{a}_2\rangle)$ with max C_1 and C_2 indicates that a rule is applicable to a fact $C_1(\mathbf{a}_1, \ell) \in \mathcal{J}$ and that it produces $C_2(\mathbf{a}_2, \ell + \Omega(e))$.

It is easy to check that the value propagation graph increases monotonically during rule application in the following sense: for each OG-ground program $\mathcal{P}$ and pseudointerpretations $\mathcal{J}$ and $\mathcal{J}'$ such that $\mathcal{I} \subseteq \mathcal{I}'$ for the corresponding limit-closed interpretations $\mathcal{I}$ and $\mathcal{I}'$, we always have $V = V'$ and $E \subseteq E'$ for the graphs (V, E, Ω) and (V', E', Ω') of $\mathcal{P}$ over $\mathcal{J}$ and $\mathcal{P}$ over $\mathcal{J}'$, respectively. This guarantees the correctness of the following definition.

Definition 5.3. *An OG-ground program $\mathcal{P}$ is* stable *if, for all pseudointerpretations $\mathcal{J}$ and $\mathcal{J}'$ such that $\mathcal{I} \subseteq \mathcal{I}'$ holds for the corresponding limit-closed interpretations $\mathcal{I}$ and $\mathcal{I}'$, for the value propagation graphs (V, E, Ω) and (V, E', Ω') of $\mathcal{P}$ over $\mathcal{J}$ and of $\mathcal{P}$ over $\mathcal{J}'$, respectively, and for each edge $e \in E$, it is the case that $\Omega(e) \leq \Omega'(e)$ holds.*

Intuitively, iterative rule applications never decrease the edge weights if a program is stable. Program $\mathcal{P}_{st}$ in Example 5.1 is stable, while $\mathcal{P}'_{st}$ is not stable since $\Omega(e) = 0$ and $\Omega'(e) = -1$ for the edge $e = (\langle C_1\rangle, \langle C_2\rangle)$ in the propagation graphs (V, E, Ω) and (V, E', Ω') of $\mathcal{P}'_{st}$ over the pseudointerpretations $\{C_1(0), C(0)\}$ and $\{C_1(1), C(0)\}$, respectively.

It can be shown that a positive-weight cycle of a stable OG-ground program $\mathcal{P}$ over a pseudointerpretation $\mathcal{J}$ guarantees divergence of numeric arguments along the cycle by repeated application of the rules of $\mathcal{P}$ to $\mathcal{J}$. This allows us to compute the pseudomaterialisation in a finite number of steps by applying the rules iteratively, provided that, after each rule application, we construct the value propagation graph for the current pseudointerpretation and bump to ∞ the numeric arguments of all pseudofacts along each positive-weight cycle. It follows that the number of rule applications needed to obtain such a cycle can be polynomially bounded in the size of $\mathcal{P}$, which leads to the following lemma.

Lemma 5.4. *For each stable OG-ground program $\mathcal{P}$, the pseudomaterialisation $\mathcal{N}(\mathcal{P})$ can be computed in time exponential in $\|\mathcal{P}\|$ and polynomial in $\|\mathcal{D}\|$, where $\mathcal{D}$ is the dataset component of $\mathcal{P}$.*

Using Lemma 5.4, we can decide fact entailment for stable OG-ground programs in EXP in combined and in P in data complexity—that is, with the same complexity as for usual Datalog. We next show how to extend this result to LL-programs with negation. We first generalise the notion of stability.

Definition 5.5. *An LL-program $\mathcal{P}$ is* stable *if it can stratified as $\mathcal{P}[1], \ldots, \mathcal{P}[h]$ such that, for each $i \in [1, h]$ and for the pseudomaterialisation $\mathcal{J}_{i-1}$ of $\mathcal{P}[1] \cup \cdots \cup \mathcal{P}[i-1]$, the reduct $\mathcal{R}(\mathcal{P}[i] \cup \mathcal{J}_{i-1})$ is stable (assuming $\mathcal{J}_0 = \emptyset$ for uniformity).*

Combining Lemma 5.4 with the ideas in Theorem 4.5, we obtain the following result.

Theorem 5.6. *The fact entailment problem for stable LL-programs is* EXP*-complete in combined and* P*-complete in data complexity.*

5.2 Type-Consistent Programs

Stability identifies a large class of LL-programs for which reasoning is tractable. Unfortunately, the condition is semantic, rather than syntactic. Moreover, it is not a local condition in the sense that it cannot be verified by looking at each rule in isolation but depends on how different rules interact. Finally, checking stability of an LL-program involves computing the reduct of each stratum, which depends on the materialisation of the preceding strata

(in fact, even checking stability of an OG-ground program is coNP-hard). This motivates the following sufficient condition for stability.

DEFINITION 5.7. *An* mm-typing *of variables of an LL-rule partitions all unguarded numeric variables occurring in positive limit body literals of the rule into* max *and* min *types. Given such a typing, a numeric term is of type* max *if it is of the form*

$$s + \left(\sum_{i=1}^{v} k_i \times m_i\right) - \left(\sum_{j=1}^{w} \ell_j \times n_j\right),$$

for s a numeric term not mentioning any max *or* min *variables, nonnegative integers v and w, each m_i a* max *variable with coefficient $k_i \geq 1$, and each n_j a* min *variable with coefficient $\ell_i \geq 1$. Moreover, a numeric term is of type* min *if the same holds except that each m_i is* min *and each is n_j* max.

An LL-rule $\rho = \varphi \to \alpha$ is type-consistent *if it has a variable mm-typing with the following properties.*

- *Each numeric variable in each negative exact literal in φ is guarded.*
- *The numeric term of each* max *and each* min *atom in ρ is of type* max *and* min*, respectively.*
- *Each comparison in φ has the form $(s_1 < s_2)$ or $(s_1 \leq s_2)$, for s_1 of type* min *and s_2 of type* max.
- *If $\alpha = C_2(\mathbf{a}_2, s_2)$ is a limit atom then, for each positive limit literal $C_1(\mathbf{a}_1, s_1)$ in φ with s_2 depending on s_1, terms s_1 and s_2 have a common unguarded variable that has coefficient 1 in s_1 and does not occur in any other positive limit literals in φ, where* dependency *is defined as in Section 5.1 except that only unguarded numeric variables are taken into account.*

An LL-program is type-consistent *if all of its rules are type-consistent.*

Type-consistency can be checked rule by rule in L, and all programs in Section 3.3 are type-consistent. Furthermore, the complexity results in Theorem 5.6 apply since type-consistency implies stability.

THEOREM 5.8. *Each type-consistent LL-program is stable.*

6. RELATED WORK

The closest formalism to limit $Datalog_{\mathbb{Z}}$ is the 'monotonic programs' of Ross and Sagiv [22]. Their core fragment extends usual Datalog by predicates whose last position ranges over partially ordered *cost domains* (e.g., integers ordered by $\leq$) with associated built-in functions. They allow only for interpretations resembling our pseudointerpretations, and their programs are required to be *monotonic* in the sense that rule applications should produce only interpretations of the appropriate form and preserve the orders of all cost domains. Unfortunately, checking monotonicity is undecidable, and moreover monotonicity does not imply decidability of fact entailment. Nonetheless, there is a rich common fragment of monotonic programs and limit-linear $Datalog_{\mathbb{Z}}$ that inherits the decidability, complexity, and tractability of the latter.

Another related formalism is Datalog^{FS} proposed by Mazuran et al. [19], which extends usual Datalog with so-called *frequency support goals* and provides the formal underpinning for the DeALS system [31]. Similar to the language of Ross and Sagiv, fact entailment in Datalog^{FS} is undecidable, and no practical decidable fragment has been identified. Finding such fragments could potentially be accomplished by transferring some ideas from our work.

$Datalog_{\mathbb{Z}}$ is also closely related to constraint logic programming (CLP). Although a number of decidable CLP languages have been identified [7], none of them allow for recursive *numeric value invention*, which is an integral feature of $Datalog_{\mathbb{Z}}$ necessary to capture several examples in Section 3.3.

Finally, note that some examples from Section 3.3 implement aggregation over recursive rules. Several attempts were made to provide a generic semantics for such aggregation (including monotonic programs and Datalog^{FS} considered above in their full power). However, all these attempts yield solutions either for restricted classes of programs that are subject to strong monotonicity assumptions, use only *min* and *max* aggregate functions, or have undecidable fact entailment [6, 11, 12, 16, 20, 27].

7. CONCLUSION

We have presented several decidable fragments of $Datalog_{\mathbb{Z}}$ that can capture interesting data analysis problems. For future work, we first aim to systematically explore the ability of $Datalog_{\mathbb{Z}}$ to express aggregate functions, which are a necessary component of any practical data analytics formalism. Second, we plan to extend our results to the context of descriptive complexity [13]: we believe that the languages of semi-positive and arbitrary LL-programs capture coNP and Δ_2^{P}, respectively. Another avenue for future work is exploring practical applicability of our algorithms and their implementation in practical declarative data analysis systems.

Acknowledgements This research is supported by EPSRC projects MaSI^3, AnaLOG, ED^3 and OASIS.

8. REFERENCES

[1] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears. BOOM analytics: Exploring data-centric,

declarative programming for the cloud. In *EuroSys*, pages 223–236, 2010.
[2] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. Design and implementation of the LogicBlox system. In *SIGMOD*, pages 1371–1382, 2015.
[3] C. Beeri, S. A. Naqvi, O. Shmueli, and S. Tsur. Set constructors in a logic database language. *J. Log. Pr.*, 10(3&4):181–232, 1991.
[4] B. Chin, D. von Dincklage, V. Ercegovac, P. Hawkins, M. S. Miller, F. J. Och, C. Olston, and F. Pereira. Yedalog: Exploring knowledge at scale. In *SNAPL*, pages 63–78, 2015.
[5] D. Chistikov and C. Haase. The taming of the semi-linear set. In *ICALP*, volume 55, pages 128:1–128:13, 2016.
[6] M. P. Consens and A. O. Mendelzon. Low complexity aggregation in GraphLog and Datalog. *Th. Comp. Sci.*, 116(1):95–116, 1993.
[7] J. Cox, K. McAloon, and C. Tretkoff. Computational complexity and constraint logic programming languages. *Ann. Math. Artif. Intell.*, 5(2–4):163–189, 1992.
[8] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.
[9] J. Eisner and N. W. Filardo. Dyna: Extending datalog for modern AI. In *Datalog*, pages 181–220, 2011.
[10] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive datalog. *ACM Trans. Database Syst.*, 22(3):364–418, 1997.
[11] W. Faber, G. Pfeifer, and N. Leone. Semantics and complexity of recursive aggregates in answer set programming. *Artif. Intell.*, 175(1):278–298, 2011.
[12] S. Ganguly, S. Greco, and C. Zaniolo. Extrema predicates in deductive databases. *J. Comput. System Sci.*, 51(2):244–259, 1995.
[13] N. Immerman. *Descriptive Complexity*. Springer, 1999.
[14] M. Kaminski, B. Cuenca Grau, E. V. Kostylev, B. Motik, and I. Horrocks. Foundations of declarative data analysis using limit datalog programs. In *IJCAI*, pages 1123–1130, 2017.
[15] M. Kaminski, B. Cuenca Grau, E. V. Kostylev, B. Motik, and I. Horrocks. Stratified negation in limit Datalog programs. In *IJCAI*, pages 1875–1881, 2018.
[16] D. B. Kemp and P. J. Stuckey. Semantics of logic programs with aggregates. In *ISLP*, pages 387–401, 1991.
[17] B. T. Loo, T. Condie, M. N. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking. *Commun. ACM*, 52(11):87–95, 2009.
[18] V. Markl. Breaking the chains: On declarative data analysis and data independence in the big data era. *PVLDB*, 7(13):1730–1733, 2014.
[19] M. Mazuran, E. Serra, and C. Zaniolo. Extending the power of datalog recursion. *VLDB J.*, 22(4):471–493, 2013.
[20] I. S. Mumick, H. Pirahesh, and R. Ramakrishnan. The magic of duplicates and aggregates. In *VLDB*, pages 264–277, 1990.
[21] C. H. Papadimitriou and M. Yannakakis. A note on succinct representations of graphs. *Information and Control*, 71(3):181–185, 1986.
[22] K. A. Ross and Y. Sagiv. Monotonic aggregation in deductive databases. *J. Comput. System Sci.*, 54(1):79–97, 1997.
[23] G. Sabidussi. The centrality index of a graph. *Psychometrika*, 31(4):581–603, 1966.
[24] J. S. Schlipf. The expressive powers of the logic programming semantics. *J. Comput. System Sci.*, 51(1):64–86, 1995.
[25] J. Seo, S. Guo, and M. S. Lam. SociaLite: An efficient graph query language based on datalog. *IEEE Trans. Knowl. Data Eng.*, 27(7):1824–1837, 2015.
[26] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo. Big data analytics with datalog queries on Spark. In *SIGMOD*, pages 1135–1149, 2016.
[27] S. Sudarshan and R. Ramakrishnan. Aggregation and relevance in deductive databases. In *VLDB*, pages 501–511, 1991.
[28] A. Van Gelder. The well-founded semantics of aggregation. In *PODS*, pages 127–138, 1992.
[29] J. Wang, M. Balazinska, and D. Halperin. Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. *PVLDB*, 8(12):1542–1553, 2015.
[30] M. Yang, A. Shkapsky, and C. Zaniolo. Scaling up the performance of more powerful datalog systems on multicore machines. *VLDB J.*, 26(2):229–248, 2017.
[31] C. Zaniolo, M. Yang, A. Das, A. Shkapsky, T. Condie, and M. Interlandi. Fixpoint semantics and optimization of recursive datalog programs with aggregates. *Th. Pract. Log. Program.*, 17(5-6):1048–1065, 2017.