Convergence of Datalog over (Pre-) Semirings

Mahmoud Abo Khamis
relationalAI
USA

Hung Q. Ngo
relationalAI
USA

Reinhard Pichler*
TU Wien
Austria

Dan Suciu*
University of Washington
USA

Yisu Remy Wang*
University of Washington
USA

ABSTRACT

Recursive queries have been traditionally studied in the framework of datalog, a language that restricts recursion to monotone queries over sets, which is guaranteed to converge in polynomial time in the size of the input. But modern big data systems require recursive computations beyond the Boolean space. In this paper we study the convergence of datalog when it is interpreted over an arbitrary semiring. We consider an ordered semiring, define the semantics of a datalog program as a least fixpoint in this semiring, and study the number of steps required to reach that fixpoint, if ever. We identify algebraic properties of the semiring that correspond to certain convergence properties of datalog programs. Finally, we describe a class of ordered semirings on which one can use the semi-naive evaluation algorithm on any datalog program.

CCS CONCEPTS

• **Theory of computation → Database query languages (principles)**.

KEYWORDS

Datalog; Semirings; Fixpoint

ACM Reference Format:
Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. 2022. Convergence of Datalog over (Pre-) Semirings. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3517804.3524140

1 INTRODUCTION

Traditionally, database systems have focused on non-recursive (loop-free) queries. However, modern data processing and tensor computations require iteration and fixpoint computation. Some systems, like Spark [54], have iteration embedded natively, while others, like Tensorflow [1] are routinely run by a driver, e.g. in order to iterate the computation until convergence.

Theoretically, the database community has studied the evaluation and optimization problem for iterative programs in the context of datalog [2], a language that restricts recursion to monotone queries over sets. However, today's applications need to iterate expressions that are not monotone queries over sets. For example standard tensor operations (e.g. Einsum, convolution) are not monotone w.r.t. set inclusion. Even some purely relational problems such as computing *betweenness centrality* [19] or *all-pairs shortest paths* (APSP) [7] require algebraic computations, and thus are not monotone in the datalog-sense.

Recursive computations beyond the Boolean space lead to three major challenges. First, non-monotonicity of general aggregations leads to difficult semantic conundra [41, 42]. Second, finite convergence behavior of the computation is no longer easy to be swept under the rug (see examples below). Third, computational optimization techniques developed for datalog, such as semi-naïve evaluation [2], break beyond the confine of Boolean monotonicity.

To address the demand for more expressive computations, this paper introduces the language datalog°, pronounced *datalogo*, which allows for expressing recursive computations over general semirings. datalog° is powerful in that it can be used to express problems such as transitive closure, all-pairs shortest-paths (APSP), minimum spanning tree (MST), or computing a local minimum of a (class of) optimization problem(s). To address the challenges, we equip datalog° with a natural (least fixpoint) semantics, study convergence behavior of datalog° programs, and generalize semi-naïve evaluation to work over semirings (modulo specific assumptions). In what follows, we *informally* describe datalog° and how we took steps towards addressing the challenges.

Expressiveness. A Datalog program is a collection of (unions of) conjunctive queries, operating on relations. Analogously, a datalog° program is a collection of (sum-)sum-product queries over a (pre-) semiring, operating on S-relations.[1] An S-relation is a function from the set of tuples to a semiring S, which is a set equipped with two operations, $\oplus$ and $\otimes$, and relational algebra extends naturally to S-relations.

Example 1.1. A real-valued matrix A is an $\mathbb{R}$-relation, where each tuple $A(i, j)$ has the value a_{ij}; and, $\mathbb{R}$ denotes the sum-product semiring $(\mathbb{R}, +, \cdot, 0, 1)$. Both the objective and gradient of the *ridge linear regression* problem $\min_x J(x)$, with $J(x) = \frac{1}{2}\|Ax - b\|^2 + (\lambda/2)\|x\|^2$, are expressible in datalog°, because they are sum-sum-product queries. The gradient $\nabla J(x) = A^\top Ax - A^\top b + \lambda x$, for example, is the following sum-sum-product query:

$$\nabla(i) = \sum_j \sum_k a(k, i) \cdot a(k, j) \cdot x(j) + \sum_j (-1) \cdot a(j, i) \cdot b(j) + \lambda() \cdot x(i)$$

*Suciu and Wang were partially supported by NSF IIS 1907997 and NSF IIS 1954222. Pichler was supported by the Austrian Science Fund (FWF):P30930.

PODS '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9260-0/22/06.
https://doi.org/10.1145/3517804.3524140

[1] The notion of *K-relations* was introduced by Green et al. [27]; we prefer to call them *S-relations* in this paper where S stands for "semiring".

The gradient has the same dimensionality as x; the group by variable is i. Gradient descent is an algorithm to solve for the solution of $\nabla J(x) = 0$, or equivalently to solve for a fixpoint solution to the datalog° program $x = f(x)$ where $f(x) = \nabla J(x) + x$.

Example 1.2. The APSP problem is to compute the shortest path length $P(x, y)$ between any pair x, y of vertices in the graph, given the length $E(x, y)$ of edges in the graph. The value-space of $E(x, y)$ can be the reals $\mathbb{R}$ or the non-negative reals $\mathbb{R}_+$. The APSP problem in datalog° can be expressed very compactly as

$$P(x, y) \text{ :- } E(x, y) \oplus \bigoplus_z P(x, z) \otimes E(z, y), \tag{1}$$

where $(\oplus, \otimes) = (\min, +)$ are the "addition" and "multiplication" operators in one of the min-+ tropical semirings $\mathsf{Trop} := (\mathbb{R} \cup \{\infty\}, \min, +, \infty, 0)$, or $\mathsf{Trop}^+ := (\mathbb{R}_+ \cup \{\infty\}, \min, +, \infty, 0)$. Here $\mathbb{R}_+$ denotes the set of non-negative reals.

By changing the semiring, datalog° is able to express similar problems in exactly the same way. For example, (1) becomes transitive closure over the Boolean semiring, top p-shortest-paths over the Trop_p^+ semiring [23] (see Example 5.7), and so forth.

Semantics. It should be clear that datalog° is very powerful. Unfortunately, "*with great power comes great responsibility*". In Datalog, the least fixpoint semantics was defined w.r.t. set inclusion [2]. Generalizing to semirings, Green at al.[27] observed that, in order to define the semantics of datalog over S-relations, one needs a *partial order*, $\sqsubseteq$, because the least fixpoint is defined w.r.t. some partial order. They proposed to use semirings which are *naturally ordered*, where $x \sqsubseteq y$ is defined as $\exists z : x + z = y$. However, some important semirings are not naturally ordered. For example, $(\mathbb{R}, +, \times, 0, 1)$ is not naturally ordered, because the relation $\exists z : x + z = y$ is *not* anti-symmetric: for any $x \neq y$ there is a z where $x + z = y$ and $y + (-z) = x$. This means recursive programs over arrays, matrices, or tensors cannot be interpreted using the framework in [27].

In datalog° we *decouple* the semiring structure from the partial order. We define a *partially ordered, pre-semiring*[2], denoted by POPS, to be any pre-semiring with a partial order, where both $\oplus$ and $\otimes$ are monotone operations. The value-space of every S-relation is some partially ordered pre-semiring.

In some cases, e.g. the Booleans, the partial order of POPS is the natural one, but in other cases it is not. For example, we can define a non-standard order relation on $\mathbb{R}$ by adding a new element, $\mathbb{R}_\bot \stackrel{\text{def}}{=} \mathbb{R} \cup \{\bot\}$, and defining $\bot \sqsubseteq x$ for all x. The set $\mathbb{R}_\bot$ is called the *lifted reals*, and we can use it to define a semantics for recursive programs over vectors, matrices, or tensors. Adding $\bot$ to create a partial order is the standard approach for defining the semantics in general programming languages [48]. In logic programming, Fitting [15] proposed adding $\bot$, leading to the 3-*valued semantics* of logic programs *with* negation.

With the partial order in place, we define the semantics of a datalog° program as the least fixpoint of the immediate consequence operator. datalog° subsumes traditional datalog semantics, *and* captures the semantics of complex, recursive computations over vectors, matrices, tensors, etc.

[2]A pre-semiring is a semiring without the axiom $x \otimes 0 = 0 \otimes x = 0$ [23]; see Sec. 2.

Finite Convergence. Example 1.1 hinted at the difficulty with computing the *exact* least fixpoint solution in a general POPS, even when we know that the fixpoint exists. In practice, numerical optimization problems are often only solved approximately. This, in principle, remains true with datalog°. However, here we concentrate on ways to compute the *exact* least fixpoint solution in a *finite* number of steps. In particular, we focus on analyzing the number of iterations needed for the naïve evaluation algorithm[3] to converge.

Note that the infinite cardinality of the semiring value-space is *not* the reason why a datalog° program does not converge when iterated naïvely. For example, intuitively we know that the naïve algorithm for the APSP program (1) may not converge under Trop (due to negative cycles), but it will always converge under Trop^+. This paper makes this intuitive observation precise: we extract algebraic properties of the POPS which serve as sufficient conditions under which the naïve algorithm for datalog° converges in a finite number of steps. Furthermore, under additional assumptions, we show that the naïve algorithm converges in poly-time. These results subsume the corresponding results in traditional Datalog.

For example, our results imply that while the naïve algorithm for (1) may not converge under Trop, it converges in a linear number of steps if the semiring is Trop^+ or more generally Trop_p^+. The property satisfied by Trop^+ but not Trop is $\mathbf{1} \oplus a = \mathbf{1}$ for every element a in the value-space and $\mathbf{1}$ is the multiplicative identity. Less formally, this says $\min(a, 0) = 0$ for all $a \in \mathbb{R}_+ \cup \{\infty\}$.

Beyond the naïve algorithm, we show that the approach of generalizing Gaussian elimination to *closed* semirings [38, 47] works for datalog° under POPS too; this leads to essentially a cubic time algorithm to find a least fixpoint of a linear datalog° program.

Optimization. Semi-naïve evaluation is one of the major optimization techniques for evaluating Datalog, which we would like to generalize to datalog°. To explain the main ideas, let us consider the Boolean semiring version of (1):

$$P(x, y) \text{ :- } E(x, y) \vee \bigvee_z P(x, z) \wedge E(z, y). \tag{2}$$

After initializing $P_0(x, y) = \delta_0(x, y) = \emptyset$, at the tth iteration semi-naïve evaluation does the following:

$$\delta_t(x, y) = \left(E(x, y) \vee \bigvee_z \delta_{t-1}(x, z) \wedge E(z, y) \right) \setminus P_{t-1}(x, y) \tag{3}$$

$$P_t(x, y) = P_{t-1}(x, y) \cup \delta_t(x, y). \tag{4}$$

Furthermore, starting from iteration $t = 2$ onwards, we can simplify (3) further by *removing* the base-case $E(x, y)$ because we know $E(x, y) \subseteq P_{t-1}(x, y)$ for $t \geq 2$:

$$\delta_t(x, y) = \left(\bigvee_z \delta_{t-1}(x, z) \wedge E(z, y) \right) \setminus P_{t-1}(x, y). \tag{5}$$

Without these optimizations, we will have rederived a lot of facts in each iteration. Under datalog°, we are able to generalize the above ideas by defining an appropriate "minus" $\ominus$ operator for certain semirings that plays the role of the $\setminus$ operator in (3), and we show that for $\ominus$, both the semi-naïve evaluation step (3) and the base-case removal optimization (5) hold for general datalog° programs.

[3]Repeatedly apply the equation until a fixpoint is reached

Example 1.3 (Shortest paths)**.** The APSP problem is 1 under Trop, where the analog of (5) is:

$$\delta_t(x,y) = (\min_z \delta_{t-1}(x,z) + E(z,y)) \ominus P_{t-1}(x,y), \tag{6}$$

where the difference operator $\ominus$ is defined later in the pape (20). Intuitively, the $\ominus$ operator does what we expect: from the new shortest paths from x to z discovered in the previous iteration, we see if adding a (z,y) edge gives us a shorter path from x to y; if so, we update the shortest path from x to y.

Paper organization. Section 2 presents basic concepts on semirings and defines (sum-)sum-product queries under semirings. Section 3 studies the *stability* of vector-valued monotone functions over posets. (Stability of a function is the number of its applications until a fixpoint is reached.) This fundamental result serves as a basis for analyzing finite convergence of the naïve algorithm for datalog°. Section 4 formulates the concept of partially preordered semirings (POPS), which form value-spaces of datalog° programs. It formally defines the syntax for datalog° programs, and introduces the fixpoint semantics of datalog°. A range of possible convergence behaviors of datalog° programs is studied in Section 5. Section 6 presents a generalization of semi-naïve evaluation to datalog°.

2 BACKGROUND ON SEMIRINGS

A *pre-semiring* [23] is a tuple $\boldsymbol{S} = (S, \oplus, \otimes, 0, 1)$ where $\oplus$ and $\otimes$ are binary operators on S for which $(S, \oplus, 0)$ is a commutative monoid, $(S, \otimes, 1)$ is a monoid, and $\otimes$ distributes over $\oplus$. This paper only considers *commutative* $\oplus$ and $\otimes$ operators. When the *absorption rule* $x \otimes 0 = 0$ holds for all $x \in S$, we call $\boldsymbol{S}$ a *semiring*.[4]

Common examples are the Booleans ($\mathbb{B} \stackrel{\text{def}}{=} \{0,1\}, \vee, \wedge, 0, 1$), and sum-product semirings over natural ($\mathbb{N}, +, \times, 0, 1$) or real numbers ($\mathbb{R}, +, \times, 0, 1$). We will refer to them simply as $\mathbb{B}, \mathbb{N}$ and $\mathbb{R}$. Other useful examples were introduced in Example 1.2: Trop and Trop$^+$. We will illustrate more semirings in this paper, and also refer the reader to [23] for many more examples.

Fix a pre-semiring $\boldsymbol{S} = (S, \oplus, \otimes, 0, 1)$ and a finite domain D; for example D could be the set $[n] = \{1, \ldots, n\}$, or some finite set of identifiers. An $\boldsymbol{S}$*-relation* is a function $R : D^k \to S$, where D is a *finite* domain. D^k is called the *key-space*, S is the *value-space*, and k is the arity of R. The *type* of R is $D^k \to \boldsymbol{S}$. If $\boldsymbol{t} \in D^k$ is a tuple of constants, then we call the expression $R(\boldsymbol{t})$ a *ground atom*. Equivalently, we can view an $\boldsymbol{S}$-relation as a mapping from ground atoms to S. Fix a vocabulary $\mathcal{R} = \{R_1, R_2, \ldots\}$ of relation names, and a semiring $\boldsymbol{S}$, where each relation R_j has type $D^{k_j} \to S$.

Definition 2.1. Let $x_1, \ldots, x_p$ be a set of variables, taking values in the domain D. A *sum-product* query is an expression of the form

$$T(x_1, \ldots, x_k) \text{ :- } \bigoplus_{x_{k+1}, \ldots, x_p \in D} A_1 \otimes \cdots \otimes A_m \tag{7}$$

where each A_u is either a *relational atom*, $R_i(x_{t_1}, \ldots, x_{t_{k_i}})$, or an equality predicate, $[x_t = x_s]$; the variables $x_1, \ldots, x_k$ are called *free variables*, and the others are called *bound variables*. The body of the query (RHS of (7)) is a *sum-product expression*.

[4] Some references, e.g. [35], define a semiring without absorption.

A *sum-sum-product* query has the form:

$$Q(x_1, \ldots, x_k) \text{ :- } T_1(x_1, \ldots, x_k) \oplus \cdots \oplus T_q(x_1, \ldots, x_k) \tag{8}$$

where $T_1, T_2, \ldots, T_q$ are sum-product expressions with the same free variables $x_1, \ldots, x_k$.

When the value-space $\boldsymbol{S}$ is the Boolean semiring, then a sum-product query is a Conjunctive Query (CQ) under set semantics, and a sum-sum-product query is a Union of Conjunctive Queries (UCQ). When $\boldsymbol{S} = \mathbb{N}$, then they are a CQ or UCQ under bag semantics; and when $\boldsymbol{S} = \mathbb{R}$ then a sum-product expression is a tensor expression, sometimes called an *Einsum* [46].

The semantics of (7) is the following. The value of each ground atom $T(\boldsymbol{t})$ is defined as the (finite!) sum on the right, where we substitute the variables $x_1, \ldots, x_k$ with the constants in the tuple $\boldsymbol{t}$. The semantics of (8) is the sum of terms on the right.

The problem of computing efficiently (sum-)sum-products over semirings has been extensively studied both in the database and in the AI literature. In databases, the query optimization and evaluation problem is a special case of sum-sum-product computation over the value-space of Booleans (set semantics) or natural numbers (bag semantics). The functional aggregate queries (FAQ) framework [32] extends the formulation to queries over multiple semirings. In AI, this problem was studied by Shenoy and Schafer [49], Dechter [13], Kohlas and Wilson [35] and others. Surveys and more examples can be found in [3, 34].

Remark 2.2. The notation pair $(\oplus, \otimes)$ helps reduce confusion when we discuss semirings such as the a tropical semiring where min is $\oplus$ and + is $\otimes$. However, $\oplus$ and $\otimes$ are quite cumbersome to parse; thus, in the rest of this paper we will mostly use + and $\circ$ instead of $(\oplus, \otimes)$ to lighten the notational density. Furthermore, just as with arithmetic multiplication, we will also sometimes drop $\circ$, writing ab for $a \circ b$ for instance.

3 LEAST FIXPOINTS OVER PRODUCT SPACES

We review partially ordered sets and present two technical results that allow us to prove the convergence of datalog° programs.

Fix a *partially ordered set* (poset), $\boldsymbol{L} = (L, \sqsubseteq)$. In this paper we will assume that each poset has a minimum element $\bot$, unless otherwise specified. We denote by $\bigvee A$, or $\bigwedge A$ respectively, the least upper bound, or greatest lower bound of a set $A \subseteq L$, when it exists. A function f between two posets is called *monotone* if $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$, and is called *strict* if $f(\bot) = \bot$.

Given a monotone function $f : \boldsymbol{L} \to \boldsymbol{L}$, a *fixpoint* is an element x such that $f(x) = x$. We denote by $\mathrm{lfp}_{\boldsymbol{L}}(f)$ the *least fixpoint* of f, when it exists, and drop the subscript $\boldsymbol{L}$ when it is clear from the context. Consider the following ω-sequence:

$$f^{(0)}(\bot) \stackrel{\text{def}}{=} \bot \qquad f^{(n+1)}(\bot) \stackrel{\text{def}}{=} f(f^{(n)}(\bot)) \tag{9}$$

If x is any fixpoint of f, then $f^{(n)}(\bot) \sqsubseteq x$ (by induction on n). There are several restrictions on the poset $\boldsymbol{L}$ and/or function f that ensure that $\bigvee_n f^{(n)}(\bot)$ is the least fixpoint of f, see [12]. The most common restriction is for f to be monotone and ω-continuous: the least fixpoint exists due to Kleene's theorem. This line was studied extensively in the formal language literature [8, 37]. In the database literature, Green et al. [27] require the semiring to be naturally

ordered ($\boldsymbol{L}$ captures the order) and ω-continuous in order to give a semantics to datalog programs over semirings.

In this paper we are interested in conditions that ensure that the sequence (9) reaches a fixpoint after a *finite* number of steps, and for this reason we introduce here two alternative conditions. An *ω-chain* in a poset $\boldsymbol{L}$ is a sequence $x_0 \leq x_1 \leq \dots$ We say that the chain is finite if there exists n_0 such that $x_{n_0} = x_{n_0+1} = x_{n_0+2} = \cdots$; equivalently, $x_{n_0} = \bigvee x_n$.

Definition 3.1 (ACC)**.** A poset $\boldsymbol{L}$ satisfies the *Ascending Chain Condition*, or ACC [43], if it has *no infinite ω-chains*. The *rank* of a strictly increasing chain $x_0 < x_1 < \cdots < x_k$ is k. We say that $\boldsymbol{L}$ has rank k if every strictly increasing chain has rank $\leq k$ [50].

Definition 3.2. A monotone function f on $\boldsymbol{L}$ (i.e. $f\colon \boldsymbol{L} \to \boldsymbol{L}$) is called p-stable if $f^{(p+1)}(\bot) = f^{(p)}(\bot)$. The *stability index* of f is the minimum p for which f is p-stable. The function f is said to be *stable* if it is p-stable for some $p \geq 0$.

If $\boldsymbol{L}$ has rank k then every monotone function f is k-stable. If ACC holds, then every monotone f is stable. If a function f is p-stable, then it has a least fixpoint and $\mathsf{lfp}(f) = f^{(p)}(\bot)$.

In this paper we need to compute the least fixpoint of functions over product spaces, $\boldsymbol{L}_1 \times \boldsymbol{L}_2$, or $\boldsymbol{L}^D$ for some set D, ordered pointwise. The ACC immediately extends to products [12]:

Proposition 3.3. *Suppose posets $\boldsymbol{L}_1, \boldsymbol{L}_2, \boldsymbol{L}$ satisfy ACC. Then $\boldsymbol{L}_1 \times \boldsymbol{L}_2$ satisfies ACC. If D is finite, $\boldsymbol{L}^D$ also satisfies ACC. If $\boldsymbol{L}_1, \boldsymbol{L}_2, \boldsymbol{L}$ have ranks k_1, k_2, k then $\boldsymbol{L}_1 \times \boldsymbol{L}_2$ has rank $k_1 + k_2$ and $\boldsymbol{L}^D$ has rank $k \cdot |D|$.*

Next, we consider stability of functions over product spaces. We start by considering two posets, $\boldsymbol{L}_1, \boldsymbol{L}_2$, and two functions:

$$f : \boldsymbol{L}_1 \times \boldsymbol{L}_2 \to \boldsymbol{L}_1 \qquad\qquad g : \boldsymbol{L}_1 \times \boldsymbol{L}_2 \to \boldsymbol{L}_2$$

Denote by $h \stackrel{\text{def}}{=} (f, g) : \boldsymbol{L}_1 \times \boldsymbol{L}_2 \to \boldsymbol{L}_1 \times \boldsymbol{L}_2$ and, for any $u \in \boldsymbol{L}_1$, denote by $g_u(y) \stackrel{\text{def}}{=} g(u, y)$. We prove:

Lemma 3.4. *Assume that p, q are two numbers such that, for all $u \in \boldsymbol{L}_1$, the function g_u is q-stable, and the function $F(x) \stackrel{def}{=} f(x, g_x^{(q)}(\bot))$ is p-stable. Then the following hold:*

- *The function h has the least fixpoint $(\bar{x}, \bar{y})$, where*

$$\bar{x} \stackrel{def}{=} F^{(p)}(\bot) \qquad\qquad \bar{y} \stackrel{def}{=} g_{\bar{x}}^{(q)}(\bot) \tag{10}$$

- *Denoting $(a_n, b_n) \stackrel{def}{=} h^{(n)}(\bot, \bot)$, the following equalities hold:*

$$a_{pq+p} = \bar{x} \qquad\qquad b_{pq+q} = \bar{y} \tag{11}$$

In particular, h is $pq + \max(p, q)$-stable.

We give here the main idea of the proof, and defer the details to the appendix. The first item of the lemma can be checked by direct calculation to verify that $f(\bar{x}, \bar{y}) = \bar{x}$ and $g(\bar{x}, \bar{y}) = \bar{y}$. For the second item, we view the computation in Eq. (10) as a sequence of steps, each applying either g or f (see Fig. 1). The fixpoint x_p is reached after $pq + p$ steps[5], while the fixpoint y_{pq} is reached after $pq + p + q$ steps. The claim follows by observing that the sequence $h^{(n)}(\bot, \bot)$ grows at least as fast as the sequence in the figure.

[5]There are pq g-steps plus p f-steps.

We note that the main idea for obtaining a fixpoint in Lemma 3.4 is a general principle that is also used in Floyd-Warshall's algorithm, and in other settings [29, 38].

We generalize the lemma as follows. A *clone* [10] over n posets $\boldsymbol{L}_1, \dots, \boldsymbol{L}_n$ is a set C where (1) each element $f \in C$ is some monotone function $f : \boldsymbol{L}_{j_1} \times \cdots \times \boldsymbol{L}_{j_k} \to \boldsymbol{L}_i$, (2) C contains all projections $\boldsymbol{L}_{j_1} \times \cdots \times \boldsymbol{L}_{j_k} \to \boldsymbol{L}_{j_i}$, and (3) C is closed under composition, i.e. it contains the function $g \circ (f_1, \dots, f_k)$ whenever $f_1, \dots, f_k, g \in C$ and their types make the composition correct. We call C a *c-clone* if it also contains the constant $\bot$ functions: $\bot\colon () \to \boldsymbol{L}_i$. For a simple illustration, if $f\colon \boldsymbol{L}_1 \times \boldsymbol{L}_2 \to \boldsymbol{L}_1$ and $g\colon \boldsymbol{L}_1 \times \boldsymbol{L}_2 \to \boldsymbol{L}_2$ are in C, then the functions $f(x, g(x, y))$, $f(f(x, y), g(x, y))$, $g(f(x, y), y)$ are also in C. Similarly, the functions $g(x, \bot)$, $g(x, g(x, \bot))$, and, in general, all functions $x \mapsto g_x^{(q)}(\bot)$ for $q \geq 0$ are in C, and so is $F(x) \stackrel{\text{def}}{=} f(x, g_x^{(q)}(\bot))$.

Theorem 3.5. *Let C be a c-clone of functions over n posets $\boldsymbol{L}_1, \dots, \boldsymbol{L}_n$, and assume that, for every $i \in [n]$, every function $f : \boldsymbol{L}_i \to \boldsymbol{L}_i$ in C is p_i-stable. WLOG, assume $p_1 \geq p_2 \geq \cdots \geq p_n$. Then, if $f_1, \dots, f_n$ are functions in C of types $f_i : \prod_{j=1,n} \boldsymbol{L}_j \to \boldsymbol{L}_i$, the function $h \stackrel{def}{=} (f_1, \dots, f_n)$ is P-stable where $P := \sum_{k=1}^{n} \prod_{i=1}^{k} p_i$. Moreover, this upper bound is tight: there exist posets $\boldsymbol{L}_1, \dots, \boldsymbol{L}_n$, a c-clone C, and functions $f_1, \dots, f_n$, such that P is the stability index of h.*

4 DATALOG°

We introduce here a new recursive language obtained by combining traditional datalog with tensor expressions. We call the language datalog°, pronounced *datalogo*, where the superscript "∘" is a (semi)-ring. We restrict our discussion to the *basic* datalog°, which is analogous to the monotone fragment of datalog without interpreted functions, and discuss extensions in Appendix B.1. Throughout this section we denote by $E_1, \dots, E_m$ the relational symbols representing the EDB predicates, by $R_1, \dots, R_n$ the symbols representing the IDB predicates.[6] All EDBs and IDBs are assumed to be over the same finite domain D, and have the same value-space $\boldsymbol{S}$; thus, their types are $E_j : D^{\ell_j} \to \boldsymbol{S}$ and $R_i : D^{k_i} \to \boldsymbol{S}$, where ℓ_j, k_i are the arities of E_j, R_i respectively.

A datalog° program P consists of n rules, one for each IDB predicate:

$$P: \qquad R_i(\mathtt{vars}_i) \text{ :- } f_i(E_1, \dots, E_m, R_1, \dots, R_n) \qquad i \in [n] \tag{12}$$

where each function $f_1, \dots, f_n$ is a sum-sum-product expression given by Def. 2.1, Eq. (8). P is called *linear* if each product contains at most one IDB predicate. The datalog° program in Ex. 1.2 is linear.

While in standard datalog rules are required to be *safe*, meaning that every variable must occur in some relational atom, we do not require this in datalog°, because we assume the domain D is finite.

Partially Ordered Pre-semirings. To define the semantics of datalog°, we extend the pre-semiring $\boldsymbol{S}$ with a partial order.

Definition 4.1. A *partially ordered pre-semiring*, in short POPS[7], is a tuple $\boldsymbol{S} = (S, +, \circ, 0, 1, \sqsubseteq)$, where $(S, +, \circ, 0, 1)$ is a pre-semiring, $(S, \sqsubseteq)$ is a poset, and $+, \circ$ are monotone.

[6]EDB and IDB stand for extensional database and intentional database respectively [2].
[7]For Partially Ordered Pre-Semiring.

A POPS satisfies the identities $\bot + \bot = \bot$ (because $\bot + \bot \sqsubseteq \bot + 0 = \bot$) and similarly $\bot \circ \bot = \bot$. We say that $\circ$ is *strict* if the identity $x \circ \bot = \bot$ holds (recall that we assume $\circ$ is commutative).

In any (pre-)semiring $\mathbb{S}$, the relation $x \preceq_S y$ defined as $\exists z : x + z = y$, is a *preorder*: it is reflexive and transitive, but not always anti-symmetric. When $\preceq_S$ is anti-symmetric, it is a partial order, in which case it is called the *natural order* on $\mathbb{S}$. Every naturally ordered semiring is a POPS, where $\bot = 0$ and $\circ$ is strict, but not conversely. Prior semantics to datalog [27] or fixpoint logic [11] required the semiring to be naturally ordered, *ω-complete*, meaning that every ω-chain has a least upper bound, and *ω-continuous*, meaning every countable sum is associative and commutative, and $\circ$ distributes over countable sums. Our approach differs in that we decouple the order relation from the (pre-)semiring structure. The reason is that many datalog° programs are over a semiring that is not naturally ordered, and, even when it is, their fixpoint semantics is over an order different from the natural one. For example, $\mathbb{R}$ is not naturally ordered, yet all tensor operations are over $\mathbb{R}$-relations, hence we need a means to interpret recursive programs over $\mathbb{R}$. The common approach for defining recursive functions over $\mathbb{R}$ or $\mathbb{N}$ (or any other set) is to "lift" the set. For example, the lifted integers $\mathbb{N}_\bot \stackrel{\text{def}}{=} \mathbb{N} \cup \{\bot\}$ with the ordering $\bot \sqsubseteq x$ for all x, is used to give a standard textbook semantics to the recursive definition of factorial:

$$P: \quad \texttt{fact}(x) = \texttt{ if } x = 0 \texttt{ then 1 else } x \times \texttt{fact}(x-1)$$

The immediate consequence operator $P(\texttt{fact})$ takes a partial function and returns a new partial function. After q iterations, $P^{(q)}(\bot)$ returns a function that is $= x!$ for $x < q$, and is $= \bot$ for $x \geq q$, and $\texttt{fact} = \mathsf{lfp}(P)$. Important for our discussion is the fact that the partial order $\sqsubseteq$ is not the natural order $x \leq y$ on $\mathbb{N}$, but a different one. This justifies our definition of a POPS (Def 4.1), where we decoupled the partial order from the (pre-)semiring operations.

Examples of POPS and discussions are given in Appendix B. It turns out that POPS capture a big variety of settings, including Fitting's 3-valued and 4-valued approaches to logic programming. The latter connection is worked out in some detail in [31].

Semantics. We define now the semantics of a basic program (12). Denote by $\boldsymbol{D}_{\text{EDB}} \stackrel{\text{def}}{=} \bigcup_{j=1,m} D^{\ell_j}$ and $\boldsymbol{D}_{\text{IDB}} \stackrel{\text{def}}{=} \bigcup_{i=1,n} D^{k_i}$, where the union is a disjoint union. An instance of all EDB predicates is $\boldsymbol{E} \in S^{\boldsymbol{D}_{\text{EDB}}}$, and an instance of all IDB predicates is $\boldsymbol{R} \in S^{\boldsymbol{D}_{\text{IDB}}}$. For each datalog° rule, the expression f_i in (12) defines a function:

$$f_i : S^{\boldsymbol{D}_{\text{EDB}}} \times S^{\boldsymbol{D}_{\text{IDB}}} \to S^{D^{k_i}}$$

Denote by $H \stackrel{\text{def}}{=} (f_1, \ldots, f_n) : S^{\boldsymbol{D}_{\text{EDB}}} \times \mathbb{S}^{\boldsymbol{D}_{\text{IDB}}} \to S^{\boldsymbol{D}_{\text{IDB}}}$.

Definition 4.2. Fix an EDB instance, $\boldsymbol{E}$. The *immediate consequence operator* of the program P in (12) is the function $F_P : S^{\boldsymbol{D}_{\text{IDB}}} \to S^{\boldsymbol{D}_{\text{IDB}}}$ defined as $F_P(\boldsymbol{R}) \stackrel{\text{def}}{=} H(\boldsymbol{E}, \boldsymbol{R})$. The *semantics* of the datalog program P is its least fixpoint, $\mathsf{lfp}(F_P)$. The *naïve evaluation algorithm* consists of setting $\boldsymbol{R}^{(0)} \stackrel{\text{def}}{=} (\bot, \ldots, \bot)$ and repeatedly computing $\boldsymbol{R}^{(q+1)} \stackrel{\text{def}}{=} F_P(\boldsymbol{R}^{(q)})$, until $\boldsymbol{R}^{(q+1)} = \boldsymbol{R}^{(q)}$. If such a fixpoint exists, then we say that the program *converges*, otherwise it *diverges*.

The Grounded datalog° **Program.** It is often more convenient to define the semantics of a datalog° program with its *grounded* version. Fix a program P as defined in Eq. (12), and consider a rule $R_i(x_1, \ldots, x_{k_i}) \text{ :- } \cdots$. A *grounding* of this rule is obtained by choosing a tuple $\boldsymbol{t} \in D^{k_i}$, substituting all head variables with the constants in $\boldsymbol{t}$, and expanding each summation operator into an explicit sum of expressions, for example $\sum_{x \in D} f(x)$ becomes $f(u_1) + f(u_2) + \cdots + f(u_{|D|})$, if the domain is $D = \{u_1, \ldots, u_{|D|}\}$. This results in an expression consisting of ground EDB atoms and ground IDB atoms connected by the operators + and $\circ$.

Define $M \stackrel{\text{def}}{=} |\boldsymbol{D}_{\text{EDB}}|$ and $N \stackrel{\text{def}}{=} |\boldsymbol{D}_{\text{IDB}}|$ the number of ground EDB atoms and ground IDB atoms respectively. Denote these atoms by the variables $e_1, \ldots, e_M$ and $x_1, \ldots, x_N$ respectively. For example, one variable x_p may stand for the ground atom $R_i(u_1, \ldots, u_{k_i})$, where $u_1, \ldots, u_{k_i} \in D$. The grounded function f_i is a multivariate polynomial in the variables $e_1, \ldots, e_M, x_1, \ldots, x_N$ over the POPS $\mathbb{S}$; $f_i : \mathbb{S}^M \times \mathbb{S}^N \to \mathbb{S}$. If the original program was linear, then each f_i is a linear function in the IDB variables $x_1, \ldots, x_N$. The *grounded* program P^g consists of all groundings of all rules in P:

$$P^g: \qquad x_k \text{ :- } f_k(e_1, \ldots, e_M, x_1, \ldots, x_N) \qquad k \in [N] \tag{13}$$

EDB and IDB instances of the original program and the grounded program are the same. For example an instance of R_i in the original program is an $\mathbb{S}$-relation, i.e. an element in $\mathbb{S}^{D^{k_i}}$, while in the grounded program R_i is replaced by $|D|^{k_i}$ variables $x_1, x_2, \ldots, x_{D^{k_i}}$, which also represent an element in $\mathbb{S}^{D^{k_i}}$. Similarly, the immediate consequence operator of the original and of the grounded program coincide. In the rest of this paper we will assume w.l.o.g. that the semantics of a datalog° program (Def. 4.2) is the fixpoint of a system of polynomials (13).

For a simple illustration, assuming a domain of size 2, $D = \{1, 2\}$, the grounding of the datalog° program in (1) is the following:

$$x_{11} \text{ :- } e_{11} + e_{11}x_{11} + e_{12}x_{21} \qquad x_{12} \text{ :- } e_{12} + e_{11}x_{12} + e_{12}x_{22}$$
$$x_{21} \text{ :- } e_{21} + e_{21}x_{11} + e_{22}x_{21} \qquad x_{22} \text{ :- } e_{22} + e_{21}x_{12} + e_{22}x_{22}$$

5 CONVERGENCE BEHAVIOR OF datalog°

In traditional datalog, the naïve evaluation algorithm converges in a number of iterations that is polynomial in the size of the domain[8], but in datalog°, the picture is more complex. Recall that D denotes the (finite) domain on which the key-space is based, which is different from the semiring value-space which can be infinite. Depending on the POPS $\mathbb{S}$, and assuming $\oplus$ and $\otimes$ take $O(1)$-time to evaluate, there are four possibilities for datalog°:

(i) Every program converges in time polynomial in $|D|$.
(ii) Every program converges in some time $T(|D|)$.
(iii) Every program converges.
(iv) Not every program converges.

We study the following question: *Given a POPS $\mathbb{S}$, determine which of the four cases holds for* datalog°?

The simplest example is when $\mathbb{S}$ is the Boolean semiring. This is the traditional datalog setup when Case (i) holds. This section provides a series of results pertaining to the question above in the

[8] In standard datalog the domain is sometimes assumed to be infinite, and instead the runtime is described in terms of the size of the *active domain*. In this paper we assume that the domain is finite, and avoid the need to talk about the active domain. When the domain is infinite, then one can adopt the approach of standard datalog and consider the active domain instead.

general case. Throughout this section we will refer the following single-rule datalog° program:

$$X() \text{ :- } A() \circ X() \quad + \quad B() \tag{14}$$

Obviously, a necessary condition for every datalog° program to converge (in one of the convergence criteria above) is for (14) to also converge. Surprisingly, in some cases the converse also holds: convergence of (14) implies convergence of any datalog° program.

A *linear function*[9] on the POPS $\mathbb{S}$ is either a constant function $f(x) \stackrel{\text{def}}{=} b$, or is defined as[10]:

$$f(x) \stackrel{\text{def}}{=} ax + b \tag{15}$$

When $b = 1$ then we say that the linear function (15) is *simple*. A multivariate function $f : \mathbb{S}^k \to \mathbb{S}$ is called *linear* if it is linear in each argument; equivalently, $f(x_1, \ldots, x_k) = a_1 \circ x_{i_1} + \cdots + a_\ell \circ x_{i_\ell} + b$, for $1 \le i_1 < i_2 < \cdots < i_\ell \le k$.

The following lemma can be verified straightforwardly; recall that $\circ$ is strict means $a \circ \bot = \bot$.

Lemma 5.1. *Let $\mathbb{S}$ be a POPS where $\circ$ is strict, and f be a univariate linear function. If $f(x) = b$, then $f^{(q)}(\bot) = b$ for all $q \ge 1$. If $f(x) = ax + b$, then, define $g(x) \stackrel{def}{=} ax + 1$, we have*

$$f^{(q)}(\bot) = b + ab + a^2b + \cdots + a^{q-1}b + \bot = g^{(q)}(\bot) \circ b. \tag{16}$$

In particular, if every simple linear function is q-stable, then every linear function is $\max(1, q)$-stable.

5.1 Divergence

It is easy to find examples where programs diverge, thus, Case (iv) is possible. Consider the program (14) over the naturally ordered semiring $\mathbb{N}$. Set $A() = 2$, $B() = 1$, then $X^{(q+1)} = 2X^{(q)} + 1$ and, after q iterations, the IDB is $X^{(q)} = 2^q - 1$, hence the program diverges.

5.2 Convergence

This section studies conditions under which a datalog° program converges in a finite number of steps that is not only dependent on $|D|$ but also on the value-space of $\mathbb{S}$. This corresponds to Case (iii). The first observation is a simple sufficient condition, which follows immediately from Proposition 3.3:

Theorem 5.2. *If ACC holds for the POPS $\mathbb{S}$, then every* datalog° *program P on $\mathbb{S}$ converges on every input EDB instance. In addition, if $\mathbb{S}$ has rank k, then the program converges in at most $k \cdot N = k \cdot |D|^{O(1)}$ iterations (where N is the number of ground IDB atoms).*

Beyond the above simple result, we give next a complete characterization of POPS $\mathbb{S}$ for which every *linear* program converges.

Theorem 5.3. *Let $\mathbb{S}$ be a POPS where $\circ$ is strict. Then, every linear* datalog° *program converges in a finite number of steps if and only if every linear function is stable.*

In [31] we give an example of a POPS $\mathbb{S}$ that belongs to Case (iii), but not to Case (ii): every datalog° program converges in a finite number of steps, yet the program P in (14) requires a number of steps that depends on the values of $A()$ and $B()$.

[9] Strictly speaking, these are *affine* functions; however, we use "linear" to align with "linear datalog".

[10] $f(x) = b$ is not a special case of (15), because, in general, $0 \circ x \neq 0$.

5.3 Convergence in $T(|D|)$ steps

Next, we study conditions under which every program converges in $T(|D|)$ steps, for some function T. Unlike the previous case, here we insist that the number of steps depends only on the size of the domain, and not on the values in the value-space. This is precisely Case (ii) in our discussion. We will give a complete characterization of the POPS $\mathbb{S}$ for this case, assuming that $\circ$ is strict.

Definition 5.4. A POPS $\mathbb{S}$ is called *p-stable* if every simple linear function is $p + 1$-stable.

A necessary condition for every program to converge in time $T(|D|)$ is that the POPS $\mathbb{S}$ is p-stable, for some $p \ge 0$. This follows immediately by considering the program (14): it converges in some fixed number of steps, say $p+1$ steps[11], which is independent of the values $A()$ and $B()$. Then, by setting $A() = a$ and $B() = 1$, it follows that the a simple linear function f is $p + 1$-stable, in particular, the POPS is p-stable. In this section we prove that p-stability of the POPS $\mathbb{S}$ is also a sufficient condition.

Theorem 5.5. *Let $\mathbb{S}$ be a p-stable POPS where $\circ$ is strict. Consider a* datalog° *program P, and recall that N is the number of ground IDB atoms. Then:*

(i) If $p = 0$, then P converges in at most N steps.
(ii) If P is linear, then it converges in $\le \sum_{i=1}^{N} (p+1)^i$ steps.
(iii) In general, P converges in $\le \sum_{i=1}^{N} (p+2)^i$ steps.

The bounds given by the theorem are exponential in N, except for 0-stable POPS. For linear programs (case (ii)), we will see in Theorem 5.11 that P can be computed in PTIME, using a different algorithm than naïve evaluation. For case (iii), we could neither prove nor disprove a polynomial bound.

In the rest of this section we outline the key steps in the proof of Theorem 5.5. The main line of development is to start from a notion of stability of a single element in the value-space, which is essentially equivalent to the stability of simple linear functions over the same space. Then, the main technical result for the proof is to bound the stability of single-valued polynomial functions using the stability of simple linear functions. This result is stated in Theorem 5.8 below. It is a statement about properties of semirings, which should be of independent interest from datalog°. The theorem is then extended to single-valued polynomial functions over stable POPS, and finally to vector-valued functions over stable POPS, which leads to Theorem 5.5.

p-Stable Semirings. We start with the value-space whose elements are stable. The following notations and simple facts can be found in [23]. Fix a semiring $\mathbb{S}$. For every $a \in \mathbb{S}$ and $p \ge 0$ define:

$$a^{(p)} \stackrel{\text{def}}{=} 1 + a + a^2 + \cdots + a^p \tag{17}$$

Definition 5.6. An element a is *p-stable* if $a^{(p)} = a^{(p+1)}$. A semiring $\mathbb{S}$ is *p-stable* if all its elements are p-stable.

This definition is consistent with the definition of POPS stability (Defn. 5.4), for the following reasons. Suppose every element of $\mathbb{S}$ is is p-stable. It is known [23] that the stability of 1 implies $\mathbb{S}$ is naturally ordered (see also [31]). Hence, $\mathbb{S}$ is a POPS using its natural

[11] The non-recursive program $X() \text{ :- } B()$ requires at least one step; this justifies writing the number of steps as $p + 1$.

order. Furthermore, one can check that every simple linear function is $p+1$-stable. Conversely, suppose $\mathbb{S}$ is naturally ordered and p-stable according to Def. 5.4, then every element a is p-stable, because $a^{(p)} = f^{(p+1)}(0)$ where f is the simple linear function. We give an example of a very useful p-stable semiring; See reference [23] for more examples.

Example 5.7 (Trop_p)**.** Consider the following semiring, which we denote by Trop_p. Its elements are bags of $p+1$ real numbers $\boldsymbol{u} = \{u_0, u_1, \ldots, u_p\} \subseteq \mathbb{R} \cup \{\infty\}$. The operations are $\boldsymbol{u} \oplus \boldsymbol{v} \stackrel{\text{def}}{=} \min_p(\boldsymbol{u} \cup \boldsymbol{v})$ and $\boldsymbol{u} \otimes \boldsymbol{v} \stackrel{\text{def}}{=} \min_p\{u_i + v_j \mid u_i \in \boldsymbol{u}, v_j \in \boldsymbol{v}\}$, where $\min_p(B)$ returns the smallest $p+1$ elements of a bag B; for example, $\min_2(\{3,5,5,5,9,9\}) = \{3,5,5\}$. The units are $0 = \{\infty, \ldots, \infty\}$ and $1 = \{0, \infty, \ldots, \infty\}$ (bags of $p+1$ elements). One can check that Trop_p is p-stable. When the relations E, P in the simple datalog$^\circ$ program (1) are interpreted over Trop_p, then the program computes the values $P(x,y)$ of the $p+1$ lowest cost paths from x to y in the graph. When $p=0$, $\mathrm{Trop}_0 = \mathrm{Trop}$ is the standard tropical semiring.

Our main technical result transfers stability from simple linear functions to *polynomial* functions:

$$f(x) \stackrel{\text{def}}{=} a_0 + a_1 x + a_2 x^2 + \cdots + a_k x^k \tag{18}$$

Theorem 5.8. *Let $\mathbb{S}$ be a p-stable commutative semiring and let f be a polynomial function* (18)*. Then, if $p=0$ then f is 1-stable; If f is linear, then it is $p+1$-stable; In general, f is $p+2$-stable.*

Abo Khamis et.al. [31] contains the proof of the above theorem, and an example explaining why in general f is not $p+1$ stable.

Theorem 5.8 generalizes special cases studied by Gondran [22, 23]. The case of a 0-stable semiring has been most extensively studied. Such semirings are called *simple* by Lehmann [38], are called *c-semirings* by Kohlas [35], and *absorptive* by Dannert et al. [11]. In all cases the authors require $1 + a = 1$ for all a (or, equivalently, $b + ab = b$ for all a, b [11]), which is equivalent to stating that a is 0-stable, and also equivalent to stating that $(\mathbb{S}, +)$ is a join-semilattice with maximal element 1. The tropical semiring is such an example; every distributive lattice is also a 0-stable semiring where we set $+ = \vee$ and $\circ = \wedge$.

The rest of the proof. To complete the proof of Theorem 5.5, we extend Theorem 5.8 to stable POPS (see [31]), and then to multi-valued polynomials (see [31]). These corollaries, stated and proved in the appendix, directly imply Theorem 5.5.

5.4 Convergence in PTIME

Assume, as before, a p-stable POPS where $\circ$ is strict. Theorem 5.5 gives an exponential (in the domain size) upper bound on the number of steps of the naive evaluation algorithm. Is this bound tight? Or can a datalog$^\circ$ program be computed in PTIME? In this section we answer partially these questions for *linear* datalog$^\circ$ programs. In this case, the immediate consequence operator is a linear function $F : \mathbb{S}^N \to \mathbb{S}^N$.

First, we consider the case of a p-stable semiring. In this case the immediate consequence operator can be written in matrix notation $F(X) = AX + B$, where A is an $N \times N$ matrix, and X, B are N-dimensional column vectors. After $q+1$ iterations, the naive algorithm computes $F^{(q+1)}(0) = B + AB + A^2B + \cdots + A^qB = A^{(q)}B$, where $A^{(q)} \stackrel{\text{def}}{=} I_N + A + A^2 + \cdots + A^q$. The naive algorithm converges in $q+1$ steps iff F is $q+1$-stable. A matrix A is called q-stable [23] if $A^{(q)} = A^{(q+1)}$. The following is easy to check: the matrix A is q-stable iff, for every vector B, the linear function $F(X) = AX + B$ is $q+1$ stable. Our discussion implies that, in order to determine the runtime of the naive algorithm on a linear datalog$^\circ$ program over a p-stable semiring, one has to compute the stability index of an $N \times N$ matrix A over that semiring. Surprisingly, no polynomial bound for the stability index of a matrix is known in general, except for $p=0$, in which case A is N-stable (see [22], and also [31]). We prove here a result in the special case when the semiring is Trop_p (introduced in Example 5.7), which is p-stable.

Lemma 5.9. *Every $N \times N$ matrix over Trop_p semiring is $(pN+p-1)$-stable. This bound is tight, i.e., there exist $N \times N$-matrices over Trop_p whose stability index is $(pN+p-1)$.*

We give the full proof in [31] and sketch here the main idea. We consider an $N \times N$-matrix over Trop_p as the adjacency matrix of a directed graph with N vertices and up to $p+1$ parallel edges from some vertex i to j. Then $A^{(k)}$ contains in position (i,j) the $p+1$ lowest-cost paths of up to k edges from i to j. The upper bound is proved by arguing that $A^{(k)}$ with $k > pN+p-1$ cannot contribute to the $p+1$ lowest-cost paths due to the existence of at least $p+1$ cycles along such paths. For proving the lower bound, we take A as the adjacency matrix of the directed N-cycle.

The lemma immediately implies:

Corollary 5.10. *Any linear* datalog$^\circ$ *program over Trop_p, converges in $pN+p-1$ steps, where $N = |D|^{O(1)}$ is the number of ground IDB tuples. This bound is tight.*

Second, we consider arbitrary p-stable POPS $\mathbb{S}$, where $\circ$ is strict, and prove that every linear datalog$^\circ$ program can be computed in PTIME, but using an algorithm different from the naïve algorithm.

When $\mathbb{S}$ is a *semiring*, then this follows from adapting Gaussian elimination to semirings [38, 47], which coincides with the Floyd-Warshall-Kleene algorithm. The algorithm and its variants compute the closure A^* of an $N \times N$ matrix in $O(N^3)$ time, in a closed semiring. The same principle applies to a p-stable POPS:

Theorem 5.11. *Let $\mathbb{S}$ be a p-stable POPS, where $\circ$ is strict, and let $f_1, \ldots, f_N$ be N linear functions in N variables. Then, there is an algorithm computing $\mathsf{lfp}(f_1, \ldots, f_N)$ in time $O(pN + N^3)$.*

6 SEMI-NAIVE OPTIMIZATION

Consider a single datalog$^\circ$ stratum (see, e.g., [2] for a definition of this standard concept in datalog) that repeatedly computes $R = F(R)$ until a fixpoint is reached. A naive evaluation consists of repeatedly applying the immediate consequence operator, that is we compute $R_0, R_1, R_2, \ldots$ where $R_{t+1} \stackrel{\text{def}}{=} F(R_t)$. As observed in standard datalog, this strategy is inefficient because all facts discovered at iteration t will be re-discovered at iterations $t+1, t+2, \ldots$ The *semi-naive* optimization consists of a modified program that computes R_{t+1} by first computing only the "novel" facts $\delta_{t+1} = F(R_t) - R_t$, which are then added to R_t to form R_{t+1}. Efficiently computing $F(R_t) - R_t$ *without* fully evaluating $F(R_t)$ is the incremental computation problem typical of incremental view maintenance. There are different

Algorithm 1: Semi-naïve evaluation for datalog°

$R_0 \leftarrow \mathbf{0}$ $\delta_0 \leftarrow \mathbf{0}$;
for $t \leftarrow 1$ **to** ∞ **do**
  $\delta_t \leftarrow F(R_{t-1}) - R_{t-1}$; // incremental computation
  $R_t \leftarrow R_{t-1} + \delta_t$;
  if $\delta_t = \mathbf{0}$ **then**
    Break
return R_t

incremental computation strategies, one of which is to exploit the fact that F is essentially multilinear (see definition below) to incrementally compute $\delta_{t+1} = F(R_{t-1} + \delta_t) - F(R_{t-1})$ without fully evaluating $F(R_t)$. This section generalizes semi-naive evaluation to datalog°.

In standard datalog the difference operator is well defined, since the Boolean semiring supports a difference operator as $x - y \stackrel{\text{def}}{=} x \wedge \neg y$. For a semi-naive evaluation in datalog°, we need to define the difference operator.

A *dioid* is a semiring $(S, +, \circ, 0, 1)$ for which $+$ is idempotent. It is known [28] that a dioid is naturally ordered. Furthermore, the natural order can be simplified by defining $a \sqsubseteq b$ iff $a + b = b$; and, under this natural order, $+$ is the same as $\vee$. (See Appendix C.1.) Dioids have many applications in a wide range of areas; see [28] for many examples.

Definition 6.1. A POPS $\mathbb{S} = (S, +, \circ, 0, 1, \sqsubseteq)$ is called a *distributive dioid* if $(S, +, \circ, 0, 1)$ is a dioid, $\sqsubseteq$ is the dioid's natural order, and the lattice is distributive. In a distributive dioid, the *difference* operator is defined by

$$b - a \stackrel{\text{def}}{=} \wedge\{c \mid a + c \sqsupseteq b\} \tag{19}$$

For example, the POPS $(2^{\boldsymbol{U}}, \cup, \cap, \emptyset, \boldsymbol{U}, \subseteq)$ is a distributive dioid, whose difference operator is exactly set-difference $b - a = \wedge\{c \mid b \subseteq a \cup c\} = b \setminus a$. The POPS Trop $= (\mathbb{R} \cup \{\infty\}, \min, +, \infty, 0, \geq)$ is a distributive dioid, whose difference operator is defined by

$$b - a = \begin{cases} b & \text{if } b \leq a \\ \infty & \text{if } b > a \end{cases} \tag{20}$$

Theorem 6.2. *Let F be a* datalog° *program over a distributive dioid, then the semi-naïve algorithm 1 returns the same solution as the naïve algorithm (Algorithm 2 in the appendix).*

A proof of the above theorem is in Appendix C. The next pillar of semi-naïve computation is the fact that the rule $F(R_{t-1}) - R_{t-1}$ can be computed efficiently by incremental computation. Note that in general F is a vector-valued second-order function, mapping input IDBs to output IDBs. We describe how to incrementally compute a component function of F. Without loss of generality, consider a component function $f(R_{t-1})$ of F. We show how

$$f(R_{t-1}) - R_{t-1} = f(R_{t-2} + \delta_{t-1}) - f(R_{t-2}) \tag{21}$$

can be computed incrementally.

Note that f is a multivariate polynomial in the input IDBs. For the purpose of incrementally computing (21) once, we can assume that no IDB in the input of f occurs twice without loss of generality; because, we can give every occurrence of an IDB a unique name. (See Example C.4 for an illustration.) Let $R_{t-2} = (A^1, \cdots, A^k)$ be the k-tuple of IDBs that f is a function of, and $\delta_{t-1} = (\delta^1, \ldots, \delta^k)$.

Theorem 6.3. *Let f be a component function of a* datalog° *program F over a distributive dioid in which every input IDB of f occurs once in the formula defining f. Then, f is multilinear, i.e.*

$$\begin{aligned} &f(A^1, \ldots, A^j + \delta^j, \ldots, A^k) \\ &= f(A^1, \ldots, A^j, \ldots, A^k) + f(A^1, \ldots, \delta^j, \ldots, A^k) \end{aligned} \tag{22}$$

If $f(\boldsymbol{A}) = c + f'(\boldsymbol{A})$ where c is a constant, then the incremental computation $f(R_{t-1}) - R_{t-1}$ can be computed by the following differential rule

$$\begin{aligned} &f(A^1 + \delta^1, \ldots, A^k + \delta^k) - f(A^1, \ldots, A^k) = \\ &\quad \sum_{j=1}^{k} f'(A^1 + \delta^1, \ldots, A^{j-1} + \delta^{j-1}, \delta^j, A^{j+1}, \ldots, A^k) \\ &\qquad - f(A^1, \ldots, A^k) \end{aligned} \tag{23}$$

The simplest application of the differential rule (23) is when F is linear, in which case the rule takes a particularly simple form:

$$\delta_{t+1} = F(R_{t-1} + \delta_t) - F(R_{t-1}) = F(\delta_t) - F(R_{t-1}) = F(\delta_t) - R_t$$

It should be noted that (23) is not the only way to implement the incremental computation. We can use (22) to expand f into up to $2^k - 1$ many terms. However, the linear expansion such as (23) is the most compact in terms of the number of rules one has to evaluate to implement the incremental computation. An example of how (23) works in the context of the APSP problem was presented in Example 1.3.

7 RELATED WORK

To empower Datalog, researchers have proposed amendments to make datalog capable of expressing some problems with greedy solutions such as APSP and MST. Most notably, the non-deterministic *choice* construct was extensively studied early on [24–26]. While datalog+choice is powerful, its expression and semantics are somewhat clunky, geared totwards answering optimization questions (greedily). In particular, it was not designed to deal with general aggregations.

To evaluate recursive datalog° program is to solve fixpoint equations over semirings, which was studied in the automata theory [37], program analysis [8, 43], and graph algorithms [6, 39, 40] communities since the 1970s. (See [23, 29, 38, 47, 55] and references thereof). The problem took slighly different forms in these domains, but at its core, it is to find a solution to the equation $x = f(x)$, where $x \in S^n$ is a vector over the domain S of a semiring, and $f : S^n \to S^n$ has multivariate polynomial component functions.

When f is affine, researchers realized that many problems in different domains are instances of the same problem, with the same underlying algebraic structure: transitive closure [53], shortest paths [18], Kleene's theorem on finite automata and regular languages [33], continuous dataflow [8, 30], etc. Furthermore, these problems share the same characteristic as the problem of computing matrix inverse [5, 21, 51]. The problem is called the *algebraic path*

problem [47], among other names, and the main task is to solve the matrix fixpoint equation $\boldsymbol{X} = \boldsymbol{A} \otimes \boldsymbol{X} \oplus \boldsymbol{I}$ over a semiring.

There are several classes of solutions to the algebraic path problem, which have pros and cons depending on what we can assume about the underlying semiring (whether or not there is a closure operator, idempotency, natural orderability, etc.). We refer the reader to [23, 47] for more detailed discussions. Here, we briefly summarize the main approaches.

The first approach is to keep iterate until a fixpoint is reached; in different contexts, this has different names: the naïve algorithm, Jacobi iteration, Gauss-Seidel iteration, or Kleene iteration. The main advantage of this approach is that it assumes less about the underlying algebraic structure: we do not need both left and right distributive law, and do not need to assume a closure operator.

The second approach is based on Gaussian elimination (also, Gauss-Jordan) elimination, which, assuming we have oracle access to the solution x^* of the 1D problem $x = 1 \oplus a \otimes x$, can solve the algebraic path problem in $O(n^3)$-time [38, 47].

The third approach is based on specifying the solutions based on the free semiring generated when viewing $\boldsymbol{A}$ as the adjanceny matrix of a graph [52]. The underlying graph structure (such as planarity) may sometimes be exploited for very efficient algorithm [39, 40].

Beyond the affine case, since the 1960s researchers in formal languages have been studying the structure of the fixpoint solution to $x = f(x)$ when f's component functions are multivariate polynomials over Kleene algebra [36, 44, 45]. It is known, for example, that Kleene iteration does not always converge (in a finite number of steps), and thus methods based on Galois connection or on widening/narrowing approaches [9] were studied. These approaches are (discrete) lattice-theoretic. More recently, a completely different approach drawing inspiration from Newton's method for solving a system of (real) equations was proposed [14, 29].

Recall that Newton's method for solving a system of equations $g(x) = 0$ over reals is to start from some point x_0, and at time t we take the first order approximation $g(x) \approx g_t(x) := g(x_t) + g'(x_t)(x - x_t)$, and set x_{t+1} to be the solution of $g_t(x) = 0$, i.e. $x_{t+1} = x_t - [g'(x_t)]^{-1} g(x_t)$. Note that in the multivariate case g' is the Jacobian, and $[g'(x_t)]^{-1}$ is to compute matrix inverse. In *beautiful* papers, Esparza et al. [14] and Hopkins and Kozen [29] were able to generalize this idea to the case when $g(x) = f(x) - x$ is defined over ω-continuous semirings. They were able to define an appropriate *minus* operator, derivatives of power series over semirings, matrix inverse, and prove that the method converges at least as fast as Kleene iteration, and there are examples where Kleene iteration does not converge, while Newton method does. Furthermore, if the semiring is commutative and idempotent (in addition to being ω-continuous), then Newton method always converges in n Newton steps. Each Newton step involves computing the Jacobian g' and computing its inverse, which is *exactly* the algebraic path problem!

In [15], Fitting proposed a three-valued semantics of logic programs with $\bot$ (= undefined) denoting "neither false (0) nor true (1)". In our terminology, this comes down to considering datalog over the partially ordered semiring THREE = $(\{\bot, \mathbf{0}, \mathbf{1}\}, \vee, \wedge, \mathbf{0}, \mathbf{1}, \leq_k)$, where $\leq_k$ denotes the *knowledge order* $\bot \leq_k \mathbf{0}, \mathbf{1}$. Negation can be introduced as a function "not" with $\texttt{not}(\mathbf{0}) = \mathbf{1}$, $\texttt{not}(\mathbf{1}) = \mathbf{0}$, and $\texttt{not}(\bot) = \bot$. The immediate consequence operator is monotone w.r.t. $\leq_k$ and thus guarantees the existence of a least fixpoint also in the presence of negation. In [16, 17], Fitting further extended his approach to Belnap's four-valued logic FOUR and, more generally, to arbitrary bilattices. FOUR $(\{\bot, \mathbf{0}, \mathbf{1}, \top\}, \leq_t, \leq_k)$ constitutes the simplest non-trivial, complete bilattice. That is, we have a complete lattice both w.r.t. the truth order $\mathbf{0} \leq_t \bot, \top \leq_t \mathbf{1}$ and w.r.t. the knowledge order $\bot \leq_k \mathbf{0}, \mathbf{1} \leq_k \top$. The additional truth value $\top$ in FOUR denotes "both false and true". It provides a means to deal with contradicting information in a meaningful way. Moreover, it guarantees the existence of a greatest fixpoint of the immediate consequence operator. Fitting uses this property to establish precise lower and upper bounds of the set of stable models in terms of the orders $\leq_t$ and $\leq_k$, with the well-founded model [20] as the smallest w.r.t. $\leq_k$. We provide a more detailed discussion of Fitting's work in [31].

Recently, semi-naïve evaluation was extended for a higher-order functional language called Datafun [4]. The book [23] contains many fundamental results on algebraic structures related to semirings and computational problems on such structures.

8 CONCLUSIONS

A massive number of application domains demand us to move beyond the confine of the Boolean world: from program analysis [8, 43], graph algorithms [6, 39, 40], provenance [27], formal language theory [37], to machine learning and linear algebra [1, 46]. Semiring and poset theory – of which POPS is an instance – is the natural bridge connecting the Boolean island to these applications.

The bridge helps enlarge the set of problems datalog$^\circ$ can express in a very natural way. The possibilities are endless. For example, amending datalog$^\circ$ with an interpretive function such as sigmoid will allow it to express typical neural network computations. Adding another semiring to the query language[12] helps express rectilinear units in modern deep learning. At the same time, the bridge facilitates the porting of analytical ideas from Datalog to analyze convergence properties of the application problems, and to carry over optimization techniques such as semi-naïve evaluation.

This paper established part of the bridge. There are many interesting open problems left open; we mention a few here.

The question of whether a datalog$^\circ$ program over p-stable POPS converges in polynomial time in p and N is open. This is open even for linear programs. Our result on Trop_p indicates that the linear case is likely in PTIME. If p-stability does not hold, then ACC was the next best barrier. It would be interesting to have a sufficient condition for convergence beyond ACC.

We can introduce negation to datalog$^\circ$ as an interpreted predicate. The question is, can we extend semantics results (such as stable model semantics) from general datalog / logic programing to datalog$^\circ$ with negation? The full version of this paper [31] contains more detailed discussions on this front. Beyond exact solution and finite convergence, as mentioned in the introduction, it is natural in some domain applications to have approximate fixpoint solutions, which will allow us to tradeoff convergence time and solution quality. A theoretical framework along this line will go a long way towards making datalog$^\circ$ deal with real machine learning, linear algebra, and optimization problems.

[12] In the functional aggregate queries [32] sense

REFERENCES

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P. A., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016* (2016), K. Keeton and T. Roscoe, Eds., USENIX Association, pp. 265–283.

[2] Abiteboul, S., Hull, R., and Vianu, V. *Foundations of Databases*. Addison-Wesley, 1995.

[3] Aji, S. M., and McEliece, R. J. The generalized distributive law. *IEEE Trans. Inf. Theory 46*, 2 (2000), 325–343.

[4] Arntzenius, M., and Krishnaswami, N. Seminaïve evaluation for a higher-order functional language. *Proc. ACM Program. Lang. 4*, POPL (2020), 22:1–22:28.

[5] Backhouse, R. C., and Carré, B. A. Regular algebra applied to path-finding problems. *J. Inst. Math. Appl. 15* (1975), 161–186.

[6] Carré, B. *Graphs and networks*. The Clarendon Press, Oxford University Press, New York, 1979. Oxford Applied Mathematics and Computing Science Series.

[7] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to algorithms*, third ed. MIT Press, Cambridge, MA, 2009.

[8] Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977* (1977), R. M. Graham, M. A. Harrison, and R. Sethi, Eds., ACM, pp. 238–252.

[9] Cousot, P., and Cousot, R. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *Programming language implementation and logic programming (Leuven, 1992)*, vol. 631 of *Lecture Notes in Comput. Sci.* Springer, Berlin, 1992, pp. 269–295.

[10] Creignou, N., Kolaitis, P. G., and Vollmer, H., Eds. *Complexity of Constraints - An Overview of Current Research Themes [Result of a Dagstuhl Seminar]* (2008), vol. 5250 of *Lecture Notes in Computer Science*, Springer.

[11] Dannert, K. M., Grädel, E., Naaf, M., and Tannen, V. Semiring provenance for fixed-point logic. In *29th EACSL Annual Conference on Computer Science Logic, CSL 2021, January 25-28, 2021, Ljubljana, Slovenia (Virtual Conference)* (2021), C. Baier and J. Goubault-Larrecq, Eds., vol. 183 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 17:1–17:22.

[12] Davey, B. A., and Priestley, H. A. *Introduction to lattices and order*. Cambridge University Press, Cambridge, 1990.

[13] Dechter, R. Bucket elimination: a unifying framework for processing hard and soft constraints. *Constraints An Int. J. 2*, 1 (1997), 51–55.

[14] Esparza, J., Kiefer, S., and Luttenberger, M. Newtonian program analysis. *J. ACM 57*, 6 (2010), 33:1–33:47.

[15] Fitting, M. A kripke-kleene semantics for logic programs. *J. Log. Program. 2*, 4 (1985), 295–312.

[16] Fitting, M. Bilattices and the semantics of logic programming. *J. Log. Program. 11*, 1&2 (1991), 91–116.

[17] Fitting, M. The family of stable models. *J. Log. Program. 17*, 2/3&4 (1993), 197–225.

[18] Floyd, R. W. Algorithm 97: Shortest path. *Commun. ACM 5*, 6 (1962), 345.

[19] Freeman, L. C. A Set of Measures of Centrality Based on Betweenness. *Sociometry 40*, 1 (Mar. 1977), 35–41.

[20] Gelder, A. V., Ross, K. A., and Schlipf, J. S. The well-founded semantics for general logic programs. *J. ACM 38*, 3 (1991), 620–650.

[21] Gondran, M. Algèbre linéaire et cheminement dans un graphe. *Rev. Française Automat. Informat. Recherche Opérationnelle Sér. Verte 9*, V-1 (1975), 77–99.

[22] Gondran, M. Les elements p-reguliers dans les dioïdes. *Discret. Math. 25*, 1 (1979), 33–39.

[23] Gondran, M., and Minoux, M. *Graphs, dioids and semirings*, vol. 41 of *Operations Research/Computer Science Interfaces Series*. Springer, New York, 2008. New models and algorithms.

[24] Greco, S., Saccà, D., and Zaniolo, C. DATALOG queries with stratified negation and choice: from P to d^P. In *Database Theory - ICDT'95, 5th International Conference, Prague, Czech Republic, January 11-13, 1995, Proceedings* (1995), G. Gottlob and M. Y. Vardi, Eds., vol. 893 of *Lecture Notes in Computer Science*, Springer, pp. 82–96.

[25] Greco, S., and Zaniolo, C. Greedy algorithms in Datalog. *Theory Pract. Log. Program. 1*, 4 (2001), 381–407.

[26] Greco, S., Zaniolo, C., and Ganguly, S. Greedy by choice. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 2-4, 1992, San Diego, California, USA* (1992), M. Y. Vardi and P. C. Kanellakis, Eds., ACM Press, pp. 105–113.

[27] Green, T. J., Karvounarakis, G., and Tannen, V. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China* (2007), L. Libkin, Ed., ACM, pp. 31–40.

[28] Gunawardena, J. An introduction to idempotency. In *Idempotency (Bristol, 1994)*, vol. 11 of *Publ. Newton Inst.* Cambridge Univ. Press, Cambridge, 1998, pp. 1–49.

[29] Hopkins, M. W., and Kozen, D. Parikh's theorem in commutative kleene algebra. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999* (1999), IEEE Computer Society, pp. 394–401.

[30] Kam, J. B., and Ullman, J. D. Global data flow analysis and iterative algorithms. *J. ACM 23*, 1 (1976), 158–171.

[31] Khamis, M. A., Ngo, H. Q., Pichler, R., Suciu, D., and Wang, Y. R. Convergence of datalog over (pre-) semirings. *CoRR abs/2105.14435* (2021).

[32] Khamis, M. A., Ngo, H. Q., and Rudra, A. FAQ: questions asked frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016* (2016), T. Milo and W. Tan, Eds., ACM, pp. 13–28.

[33] Kleene, S. C. Representation of events in nerve nets and finite automata. In *Automata studies*, Annals of mathematics studies, no. 34. Princeton University Press, Princeton, N. J., 1956, pp. 3–41.

[34] Kohlas, J. *Information algebras - generic structures for inference*. Discrete mathematics and theoretical computer science. Springer, 2003.

[35] Kohlas, J., and Wilson, N. Semiring induced valuation algebras: Exact and approximate local computation algorithms. *Artif. Intell. 172*, 11 (2008), 1360–1399.

[36] Kuich, W. The Kleene and the Parikh theorem in complete semirings. In *Automata, languages and programming (Karlsruhe, 1987)*, vol. 267 of *Lecture Notes in Comput. Sci.* Springer, Berlin, 1987, pp. 212–225.

[37] Kuich, W. Semirings and formal power series: their relevance to formal languages and automata. In *Handbook of formal languages, Vol. 1*. Springer, Berlin, 1997, pp. 609–677.

[38] Lehmann, D. J. Algebraic structures for transitive closure. *Theor. Comput. Sci. 4*, 1 (1977), 59–76.

[39] Lipton, R. J., Rose, D. J., and Tarjan, R. E. Generalized nested dissection. *SIAM J. Numer. Anal. 16*, 2 (1979), 346–358.

[40] Lipton, R. J., and Tarjan, R. E. Applications of a planar separator theorem. *SIAM J. Comput. 9*, 3 (1980), 615–627.

[41] Liu, Y. A., and Stoller, S. D. Founded semantics and constraint semantics of logic rules. *J. Log. Comput. 30*, 8 (2020), 1609–1668.

[42] Liu, Y. A., and Stoller, S. D. Recursive rules with aggregation: A simple unified semantics, 2020.

[43] Nielson, F., Nielson, H. R., and Hankin, C. *Principles of program analysis*. Springer-Verlag, Berlin, 1999.

[44] Parikh, R. J. On context-free languages. *J. Assoc. Comput. Mach. 13* (1966), 570–581.

[45] Pilling, D. L. Commutative regular equations and Parikh's theorem. *J. London Math. Soc. (2) 6* (1973), 663–666.

[46] Rocktäschel, T. Einsum is all you need - Einstein summation in deep learning. https://rockt.github.io/2018/04/30/einsum.

[47] Rote, G. Path problems in graphs. In *Computational graph theory*, vol. 7 of *Comput. Suppl.* Springer, Vienna, 1990, pp. 155–189.

[48] Scott, D., and Strachey, C. Toward a mathematical semantics for computer languages, 1971.

[49] Shenoy, P. P., and Shafer, G. Axioms for probability and belief-function proagation. In *UAI '88: Proceedings of the Fourth Annual Conference on Uncertainty in Artificial Intelligence, Minneapolis, MN, USA, July 10-12, 1988* (1988), R. D. Shachter, T. S. Levitt, L. N. Kanal, and J. F. Lemmer, Eds., North-Holland, pp. 169–198.

[50] Stanley, R. P. *Enumerative combinatorics. Volume 1*, second ed., vol. 49 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, Cambridge, 2012.

[51] Tarjan, R. E. Graph theory and gaussian elimination, 1976. J.R. Bunch and D.J. Rose, eds.

[52] Tarjan, R. E. A unified approach to path problems. *J. ACM 28*, 3 (1981), 577–593.

[53] Warshall, S. A theorem on boolean matrices. *J. ACM 9*, 1 (1962), 11–12.

[54] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., and Stoica, I. Apache spark: a unified engine for big data processing. *Commun. ACM 59*, 11 (2016), 56–65.

[55] Zimmermann, U. Linear and combinatorial optimization in ordered algebraic structures. *Ann. Discrete Math. 10* (1981), viii+380.

APPENDIX

A PROOFS AND EXTENSIONS OF SECTION 3

A.1 Proof of Lemma 3.4

Recall that, for every $a \in \boldsymbol{L}_1$, g_a denotes the function $g_a(y) = g(a, y)$; its type is $\boldsymbol{L}_2 \to \boldsymbol{L}_2$, and $g_a^{(k)} \in C$ for every $k \geq 0$, because C is a c-clone. Similarly, the type of $F(x) \stackrel{\text{def}}{=} f(x, g^{(q)}(x, \bot))$ is $\boldsymbol{L}_1 \to \boldsymbol{L}_1$, and $F^{(k)} \in C$ for all $k \geq 0$.

Claim 1. The pair $(\bar{x}, \bar{y})$ defined by (10) is a fixpoint of h.

The claim follows immediately from

$$g(\bar{x}, \bar{y}) = g_{\bar{x}}(\bar{y}) = g_{\bar{x}}(g_{\bar{x}}^{(q)}(\bot) = g_{\bar{x}}^{(q+1)}(\bot) = g_{\bar{x}}^{(q)}(\bot) = \bar{y}$$

by the definition of $\bar{y}$ and the q-stability of $g_{\bar{x}}$. Similarly,

$$f(\bar{x}, \bar{y}) = f(\bar{x}, g_{\bar{x}}^{(q)}(\bot)) = F(\bar{x}) = \bar{x}$$

by the definition of $\bar{x}$ and the p-stability of F, proving that $(\bar{x}, \bar{y})$ is a fixpoint of h.

Claim 2. The pair $(\bar{x}, \bar{y})$ defined by (10) is a *least* fixpoint of h.

Since h is monotone, it converges to the least fixpoint. It follows that, for every $n \geq 0$, $(a_n, b_n) \stackrel{\text{def}}{=} h^{(n)}(\bot, \bot) \sqsubseteq (\bar{x}, \bar{y})$. Hence, to prove Claim 2 it is sufficient to show that $(\bar{x}, \bar{y}) \sqsubseteq (a_n, b_n)$ for some n.

We will show that $(\bar{x}, \bar{y}) \sqsubseteq (a_{pq+p+q}, b_{pq+p+q})$. To prove this inequality, we observe that the pair $(\bar{x}, \bar{y})$ defined by (10) is the last term of the following (not necessarily increasing) sequence:

$$\begin{aligned}(\bot, \bot) =& (x_0, y_{0,0}), (x_0, y_{0,1}), \cdots, (x_0, y_{0,q}),\\ & (x_1, y_{1,0}), (x_1, y_{1,1}) \cdots (x_1, y_{1,q}),\\ & \cdots\\ & (x_p, y_{p,0}), (x_p, y_{p,1}), \cdots, (x_p, y_{p,q}) = (\bar{x}, \bar{y}) \end{aligned} \tag{24}$$

where:

$$x_0 \stackrel{\text{def}}{=} \bot \qquad y_{k,0} \stackrel{\text{def}}{=} \bot$$
$$x_{k+1} \stackrel{\text{def}}{=} f(x_k, y_{k,q}) \qquad y_{k,\ell+1} \stackrel{\text{def}}{=} g(x_k, y_{k,\ell})$$

In the $(x_k, y_{k,\ell})$ sequence each element is obtained from the previous one by applying either a g-step (from $(x_k, y_{k,\ell})$ to $(x_k, y_{k,\ell+1})$) or an f-step (from $(x_k, y_{k,q})$ to $(x_{k+1}, y_{k+1,0})$). The sequence (24) can be visualized as:

$$(gg \cdots g)\ f\ (gg \cdots g)\ f\ \cdots\ (gg \cdots g)\ f\ (gg \cdots g)$$

Instead of proving $(\bar{x}, \bar{y}) \sqsubseteq (a_{pq+p+q}, b_{pq+p+q})$, we prove a stronger claim, which completes the proof of Claim 2

Claim 3. For every $n \geq 0$, $(x_k, y_{k,\ell}) \sqsubseteq (a_n, b_n)$, where $n = k(q+1) + \ell$ is the position of $(x_k, y_{k,\ell})$ in the sequence (24).

Before proving the claim, we notice $h^{(n)}(\bot, \bot) \sqsubseteq h^{(n+1)}(\bot, \bot)$ implies $a_n \sqsubseteq f(a_n, b_n)$ and $b_n \sqsubseteq g(a_n, b_n)$. Now we prove the claim by induction. Assume the claim holds for $(x_k, y_{k,\ell})$. If the next element $(x_k, y_{k,\ell+1})$ is obtained via a g-step then we have:

$$\begin{aligned}(x_k, y_{k,\ell+1}) =& (x_k, g_{x_k}(y_{k,\ell})) = (x_k, g(x_k, y_{k,\ell})) \sqsubseteq (a_n, g(a_n, b_n))\\ \sqsubseteq& (f(a_n, b_n), g(a_n, b_n)) = (a_{n+1}, b_{n+1})\end{aligned}$$

hence the claim holds for $(x_k, y_{k,\ell+1})$ as well. If the next element $(x_{k+1}, y_{k+1,0})$ is obtained via an f-step (which happens when $\ell = q$) then we use the fact that $y_{k+1,0} = \bot$ and derive:

$$\begin{aligned}(x_{k+1}, \bot) =& (f(x_k, y_{k,q}), \bot) \sqsubseteq (f(x_k, y_{k,q}), y_{k,q})\\ \sqsubseteq& (f(a_n, b_n), b_n) \sqsubseteq (f(a_n, b_n), g(a_n, b_n)) = (a_{n+1}, b_{n+1})\end{aligned}$$

This completes the proof of the claim.

Finally, to prove Lemma 3.4, note that $\bar{x} = x_p$ was already reached at the beginning of the last line in (24), thus $\bar{x} = a_n$ for $n = pq + p$. By switching the roles of f and g, we can reach the same least fixpoint $(\bar{x}, \bar{y})$ using a sequence $(f \cdots f)g(f \cdots f)g \cdots g(f \cdots f)$, similar to (24). The end of this sequence is the same $(\bar{x}, \bar{y})$ because the least fixpoint is unique. By the same argument as above, we have $\bar{y} = b_n$ for $n = pq + q$. This completes the proof of the lemma.

B PROOFS AND EXTENSIONS OF SECTION 4

B.1 Extensions to datalog°

We describe here several extensions to the definition of datalog° in Sec. 4, which are necessary both in order to make the language useful for real applications, and in order to express some advanced optimization techniques, which we pursue in a different project.

First, we allow the use of arbitrary interpreted function on the key-space, and extend Definition 2.1 with a *conditional sum-sum-product* query, which has the form:

$$Q(x_1, \ldots, x_k) \text{ :- } \texttt{case } C_1 : Q_1;\ \ C_2 : Q_2; \cdots;\ \texttt{else } Q_n \tag{25}$$

where $C_1, C_2, \ldots$ are conditions on the variables $x_1, \ldots, x_k$, and $Q_1, Q_2, \ldots$ are sum-sum-product expressions over the same variables $x_1, \ldots, x_k$. For example, we may compute the prefix-sum of a vector V using the following datalog° rule:

$$W(i) \text{ :- } \texttt{case } i = 0 : V(0) \texttt{ else } W(i-1) + V(i)$$

Here $i - 1$ is an interpreted function on the key-space. All results in Section 5 continue to hold when datalog° is extended with interpreted functions over the key-space and with `case`-statements, because the grounding of each rule R_i :- f_i continues to be a polynomial over the POPS $\boldsymbol{S}$.

The next extensions are necessary to make datalog° practical, and the results in Section 5 may no longer hold. We pursue these extensions in a different project where we study optimizations for datalog°:

Infinite Key Domains We remove the restriction from Sec. 2 that requires the key domain D to be finite. Instead, we require the active domain of the EDB predicates to be finite, and require all datalog° rules to be *safe*, which guarantees that the active domain of all IDB predicates are also finite.

Multiple Domains We allow both multiple key domains, and multiple value domains (POPS). The types of all EDB and IDB predicates must be declared at the beginning of a datalog° program, for example:

```
E(String, String) : Bool;
Cost(String, String, N) : Bool;
Path(String, String) : Trop;
```

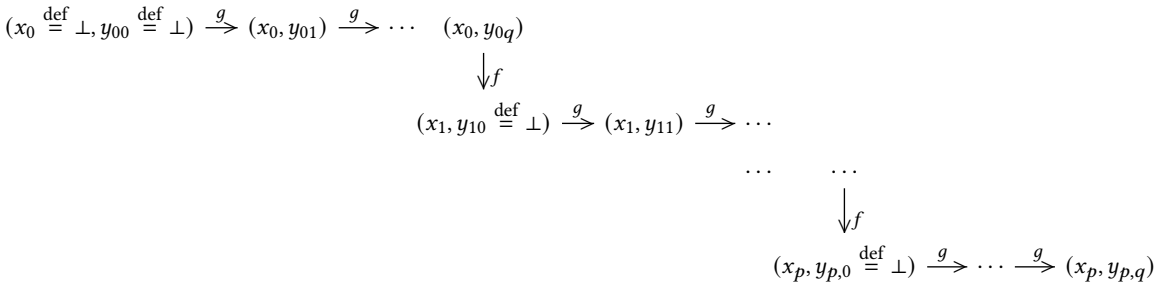

Figure 1: Computing the fixpoint of (f, g).

Cast Operator We extend datalog° with a cast operator, $[-] : \mathbb{B} \to S$, defined as follows:

$$[0] \stackrel{\text{def}}{=} \bot_S \qquad\qquad [1] \stackrel{\text{def}}{=} 1_S$$

We notice that the cast operator is monotone. When $+_S$ is idempotent, then the cast operator is also a pre-semiring homomorphism. When S is a naturally ordered semiring, then $[0] = 0_S$ and $[1] = 1_S$.

Keys to Values We allow key values to be used as POPS values, when the types are right. For example, referring to the declarations above, we may write:

$$\texttt{Path}(x, y) \text{ :- } \min_c ([\texttt{Cost}(x, y, c)] + c)$$

The expression $[\texttt{Cost}(x, y, c)]$ is a cast from Booleans to the tropical semiring, i.e. its value is ∞ or 0, and we add to it the cost c, which is of type $\mathbb{N}$, hence can be cast to Trop.

Stratification Finally, we allow the use of arbitrary interpreted functions over the value domains. Some of these functions may not be monotone; in that case we require the datalog° program to be stratified, in the usual way.

B.2 Pre-semirings to POPS

We describe a general procedure to convert a pre-semiring to a POPS.

Definition B.1. Let S be a pre-semiring. We say that a POPS S_1 *extends* S if $S \subseteq S_1$, and the operations $\oplus, \otimes, 0, 1$ in S are the same as those in S_1.

We describe ways to extend a pre-semiring $S = (S, +, *, 0, 1)$ to a POPS, all inspired by abstract interpretations in programming languages [8].

Representing Undefined The lifting operation is defined by setting $S_\bot \stackrel{\text{def}}{=} S \cup \{\bot\}$ and extending the operations to $x + y = x * y = \bot$ whenever $x = \bot$ or $y = \bot$. The value $\bot$ represents undefined.

Representing Contradiction We further extend $S_\bot$ by defining $S_\bot^\top \stackrel{\text{def}}{=} S_\bot \cup \{\top\}$ and setting $\bot + \top = \bot * \top = \bot$, $0 * \top = 0$, and $x + \top = x * \top = \top$ when $x \neq \bot, x \neq 0$. The new element $\top$ represents contradiction. Intuitively: $\bot$ is the empty set $\{\}$, while $\top$ is the entire set S.

Incomplete values More generally, define $I \stackrel{\text{def}}{=} \mathcal{P}_{\text{fin}}(S) \cup \{S\}$; that is, I consists of all finite subsets of S, plus S itself (which we add explicitly when S is infinite). The operations $\oplus, \otimes$ are defined set-wise, such that, if the result of an operation is an infinite set, then we replace it with S. Intuitively, a finite set $\{u, v, w\} \in I$ represents a value that can be either u or v or w. Here $\bot = \{\}$ and $\top = S$.

B.3 Strictness of $\oplus$ and $\otimes$ are independent

We give two examples of POPS where only one of the two operators $\oplus, \otimes$ is strict.

Strict $\otimes$ In any non-trivial naturally ordered semiring $\otimes$ is strict, while $\oplus$ is not. For example, consider $(\mathbb{N}, +, *, 0, 1, \leq)$. Then $*$ is strict because $x * 0 = 0$, while $+$ is not strict because $x + 0 \neq 0$ for $x \neq 0$.

Strict $\otimes$ Next, consider the semiring $(\mathbb{N} \cup \{\top\}, \oplus, \otimes, \{0\}, \{1\}, \supseteq)$, where $\top$ the infinite set $\{0, 1, 2, \ldots\}$. We view each element $x \in \mathbb{N}$ as the singleton set $\{x\}$, and define the operations $\oplus, \otimes$ set-wise: $A \oplus B \stackrel{\text{def}}{=} \{x + y \mid x \in A, y \in B\}$, $A \otimes B \stackrel{\text{def}}{=} \{x * y \mid x \in A, y \in B\}$, where we replace the result with $\top$ if the resulting set is not a singleton. Concretely, we have:

$$\{x\} \oplus \{y\} = \{x + y\} \qquad \{x\} \oplus \top = \top$$
$$\{x\} \otimes \{y\} = \{x * y\} \qquad \{0\} \oplus \top = \{0\} \qquad x \neq 0 : \ \{x\} \oplus \top = \top$$

Since $\top$ is the smallest element, $\oplus$ is strict, while $\otimes$ is not.

B.4 Need for Pre-Semirings

$\mathbb{N}_\bot$ and the similarly defined $\mathbb{R}_\bot$ are pre-semirings. They are not semirings, because $\bot * 0 = \bot$, thus 0 is not absorptive. It is possible to define an extension $\mathbb{N} \cup \{\bot\}$ of $\mathbb{N}$ that is a semiring, by simply re-defining $\bot * 0 = 0$ and $\bot * x = \bot$ for $x \neq 0$. However, no POPS extension of $\mathbb{R}$ exists that is a semiring:

Lemma B.1. *If S is any POPS extension of $(\mathbb{R}, +, *, 0, 1)$, then S is not a semiring, i.e. it fails the absorption law $0 * x = 0$.*

Proof. Let $S = (S, +, *, 0, 1, \sqsubseteq)$ be an ordered semiring that is an extension of the semiring $(\mathbb{R}, +, *, 0, 1)$. In particular $\mathbb{R} \subseteq S$ and S has a minimal element $\bot$. Since 0, 1 are identity elements for $+, *$ we have:

$$\bot + 0 = \bot \qquad\qquad \bot * 1 = \bot$$

We claim that the following more general identities hold:

$$\forall x \in \mathbb{R} : \ \bot + x = \bot \qquad\qquad \forall x \in \mathbb{R} - \{0\} : \bot * x = \bot$$

To prove the first identity, we use the fact that + is monotone in S and $\bot$ is the smallest element and derive $\bot + x \sqsubseteq (\bot + (y - x)) + x = \bot + y$ for fall x, y. This implies $\bot + x = \bot + y$ for all x, y and the claim follows by setting $y = 0$. The proof of the second identity is similar: first observe that $\bot * x \sqsubseteq (\bot * \frac{y}{x}) * x = \bot * y$ hence $\bot * x = \bot * y$ for all $x, y \in \mathbb{R} - \{0\}$, and the claim follows by setting $y = 1$.

Since S is a semiring it satisfies the absorption law, hence $\bot * 0 = 0$. We prove now that $0 = \bot$. Choose any $x \in \mathbb{R} - \{0\}$, and derive:

$$\bot * (x + (-x)) = \bot * 0 = 0$$
$$(\bot * x) + (\bot * (-x)) = \bot + \bot = \bot$$

By distributivity, the two lines are equal, hence $0 = \bot$, and thus 0 is the smallest element in S. Then, for every $x \in \mathbb{R}$, we have $x + 0 \sqsubseteq x + (-x) = 0$, which implies implies $x = 0$, contradiction. □

C PROOFS AND EXTENSIONS OF SECTION 6

C.1 Dioid

For completeness, we quickly prove the following

Proposition C.1. *Let $(S, +, \circ, 0, 1)$ be a dioid, i.e. a semiring where + is idempotent. Then, the following hold:*

(i) The relation $\preceq$ defined by $a \preceq b$ iff $a + b = b$ is a partial order, and it is the same as the natural order of the semiring.
(ii) + is the same as $\vee$

Proof. We first show that $\preceq$ is the natural order $\sqsubseteq$, which is defined by $a \sqsubseteq b$ iff $\exists c : a + c = b$. One direction, $a \preceq b$ implies $a \sqsubseteq b$ is obvious. For the converse, assume $a + c = b$. Then, $a + b = a + (a + c) = a + c = b$ due to idempotency, and thus $a \preceq b$. The relation $\sqsubseteq$ is a preorder; to make it a partial order we only need to verify anti-symmetry, which is easy using the $\preceq$ relation: $a + b = b$ and $a + b = a$ imply $a = b$. We just proved (i).

To show (ii), let $c = a + b$ for some a, b, c. Then $a + c = a + a + b = a + b = c$; thus, $a \sqsubseteq c$. Similarly $b \sqsubseteq c$, which means $a \vee b \sqsubseteq c = a + b$. Conversely, let $d = a \vee b$. Then, $a + d = d$ and $b + d = d$, which means $a + b + d = d$ and thus $a + b \sqsubseteq d = a \vee b$. □

Algorithm 2: Naïve evaluation for datalog°

```
R_0 ← 0;
for t ← 1 to ∞ do
    R_t ← F(R_{t-1});
    if R_t = R_{t-1} then
        Break
return R_t
```

C.2 Proof of Theorem 6.2

Two main properties of distributive dioids we need for semi-naïve evaluation of datalog° are proved in the following lemma, which is then used to prove Theorem 6.2.

Lemma C.2. *Let $S = (S, +, \circ, 0, 1, \sqsubseteq)$ be a distributive dioid, then, with the $-$ operator defined in* (19)*, for any $a, b, c \in S$,*

$$a + (b - a) = b \qquad \text{if } a \sqsubseteq b \tag{26}$$
$$(a + b) - (a + c) = b - (a + c) \tag{27}$$

Proof. To prove these identities we establish several properties of the $-$ operation in a complete PODS as defined in Eq. (19). First, we show

$$b \sqsubseteq a + (b - a) \qquad \forall a, b \in S \tag{28}$$

This follows because

$$\begin{aligned} a + (b - a) &= a + \wedge\{c \mid a + c \sqsupseteq b\} \\ &= a \vee (\wedge\{c \mid a \vee c \sqsupseteq b\}) \\ \text{(distributivity)} &= \wedge\{a \vee c \mid a \vee c \sqsupseteq b\} \\ &\sqsupseteq b \end{aligned}$$

Next, if $a \sqsubseteq b$ then $a + b = a \vee b = b$. It follows that $b \sqsupseteq \wedge\{c : a + c \sqsupseteq b\} = b - a$. Thus,

$$a \sqsubseteq b \qquad \Longrightarrow \qquad a + (b - a) \sqsubseteq a + b = b. \tag{29}$$

Together with (28), we just proved (26).

Next, we prove (27). For any d for which $a + c + d \sqsupseteq b$, $a + a + c + d \sqsupseteq a + b$ which implies $a + c + d \sqsupseteq a + b$ because + is idempotent. Conversely, $a + c + d \sqsupseteq a + b$ implies $a + c + d \sqsupseteq b$ because $a + b \sqsupseteq b$. Hence,

$$\begin{aligned} (a + b) - (a + c) &= \wedge\{d \mid a + c + d \sqsupseteq a + b\} \\ &= \wedge\{d \mid a + c + d \sqsupseteq b\} \\ &= b - (a + c) \end{aligned}$$

□

Proof of Theorem 6.2. It is easy to see that F is monotone, i.e. $X \sqsubseteq Y$ implies $F(X) \sqsubseteq F(Y)$. Hence, by induction we have $R_{t-1} = F^{(t-1)}(\bot) \sqsubseteq F^{(t)}(\bot) = R_t$ for all $t \geq 1$. Thus, from (26) we have $\delta_t + R_{t-1} = (F(R_{t-1}) - R_{t-1}) + R_{t-1} = F(R_{t-1})$. □

C.3 Proof of Theorem 6.3

Proof. From the fact that + is idempotent, $b + b = b$, we have

$$a \circ (x + \delta) + b = (a \circ x + b) + (a \circ \delta + b),$$

from which equality (22) follows.

The correctness of the differential rule follows from (22) and (27). Note that we were able to drop one $+f(A^1, \ldots, A^k)$ term thanks to (27). □

C.4 Examples

Example C.3 (Linear transitive closure)**.** For the recursion,

$$T(x, y) \text{ :- } E(x, y) \vee \exists z : T(x, z) \wedge E(z, y) \tag{30}$$

which is Boolean datalog°, the differential rule at iteration $t \geq 2$ is

$$\delta_t(x, y) = (\exists z : \delta_{t-1}(x, z) \wedge E(z, y)) \setminus T_{t-1}(x, y). \tag{31}$$

We were able to remove the base-case thanks to (23).

Example C.4 (Quadratic transitive closure)**.** For the recursion,

$$T(x, y) \text{ :- } E(x, y) \vee (\exists z : T(x, z) \wedge T(z, y)) \tag{32}$$

removing the base-case, the differential rule is

$$\delta_t(x, y) = (\exists z : T_{t-1}(x, z) \wedge \delta_{t-1}(z, y)) \tag{33}$$
$$\cup\ (\exists z : \delta_{t-1}(x, z) \wedge T_{t-2}(z, y)) \tag{34}$$
$$\setminus\ T_{t-1}(x, y). \tag{35}$$