Convergence of datalog over (Pre-) Semirings

MAHMOUD ABO KHAMIS and HUNG Q. NGO, Relational AI, Berkeley, USA
REINHARD PICHLER, TU Wien, Vienna, Austria
DAN SUCIU and YISU REMY WANG, University of Washington, Seattle, USA

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 article, 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-naïve evaluation algorithm on any datalog program.

CCS Concepts: • **Theory of computation → Database query languages (principles)**;

Additional Key Words and Phrases: Datalog, semirings, fixpoint

ACM Reference Format:
Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. 2024. Convergence of datalog over (Pre-) Semirings. *J. ACM* 71, 2, Article 8 (April 2024), 55 pages. https://doi.org/10.1145/3643027

1 INTRODUCTION

For 50 years, the relational data model has been the main choice for representing, modeling, and processing data. Its main query language, SQL, is found today in a wide range of applications and devices, from smart phones, to database servers, to distributed cloud-based clusters. The reason for its success is the *data independence principle*, which separates the declarative model from the physical implementation [10] and enables advanced implementation techniques, such as cost-based optimizations, indices, materialized views, incremental view maintenance, parallel evaluation, and many others, while keeping the same simple, declarative interface to the data unchanged.

But scientists today often need to perform tasks that require iteration over the data. Gradient descent, clustering, page-rank, network centrality, inference in knowledge graphs are some examples of common tasks in data science that require iteration. While SQL has introduced recursion

Suciu and Wang were partially supported by NSF IIS 1907997 and NSF IIS 1954222. Pichler was supported by the Austrian Science Fund (FWF):P30930 and by the Vienna Science and Technology Fund (WWTF) [10.47379/ICT2201].
Authors' addresses: M. Abo Khamis and H. Q. Ngo, Relational AI, 2120 University Ave, Berkeley, CA 94704, USA; e-mails: mahmoudabo@gmail.com, hung.q.ngo@gmail.com; R. Pichler, TU Wien, Faculty of Informatics, Institute of Logic and Computation, Database and Artificial Intelligence Group, 1040 Vienna, Austria; e-mail: pichler@dbai.tuwien.ac.at; D. Suciu and Y. R. Wang, University of Washington, Department of Computer Science and Engineering, Box 352350, Seattle, WA 98195-2350, USA; e-mails: suciu@cs.washington.edu, remywang@cs.washington.edu.

ACM 0004-5411/2024/04-ART8
https://doi.org/10.1145/3643027

since 1999 (through **Common Table Expressions (CTE)**), it has many cumbersome limitations and is little used in practice [62].

The need to support recursion in a declarative language led to the development of *datalog* in the mid '80s [80]. Datalog adds recursion to the relational query language, yet enjoys several elegant properties: It has a simple, declarative semantics; its naïve bottom-up evaluation algorithm always terminates; and it admits a few powerful optimizations, such as semi-naïve evaluation and magic set rewriting. Datalog has been extensively studied in the literature; see Reference [37] for a survey and References [60, 80] for historical notes.

However, datalog is not the answer to modern needs, because it only supports monotone queries over sets. Most tasks in data science today require the interleaving of recursion and aggregate computation. Aggregates are not monotone under set inclusion, and therefore they are not supported by the framework of pure datalog. Neither SQL'99 nor popular open-source datalog systems like Soufflé [42] allow recursive queries to have aggregates. While several proposals have been made to extend datalog with aggregation [11, 16, 26, 27, 39, 61, 71, 74, 75, 83–87], these extensions are at odds with the elegant properties of datalog and have not been adopted by either datalog systems or SQL engines.

In this article, we propose a foundation for a query language that supports both recursion and aggregation. Our proposal is based on the concept of K-relations, introduced in a seminal paper [38]. In a K-relation, tuples are mapped to a fixed semiring. Standard relations (sets) are $\mathbb{B}$-relations where tuples are mapped to the Boolean semiring $\mathbb{B}$, relations with duplicates (bags) are $\mathbb{N}$-relations, sparse tensors are $\mathbb{R}$-relations, and so on. Queries over K-relations are the familiar relational queries, where the operations $\wedge, \vee$ are replaced by the operations $\otimes, \oplus$ in the semiring; importantly, an existential quantifier $\exists$ becomes an $\oplus$-aggregate operator. K-relations are a very powerful abstraction, because they open up the possibility of adapting query processing and optimization techniques to other domains [4].

Our first contribution is to introduce an extension of datalog to K-relations. We call the language datalog$^\circ$, where the superscript o represents a (semi)-ring. datalog$^\circ$ has a declarative semantics based on the least fixpoint and supports both recursion and aggregates. We illustrate throughout this article its utility through several examples that are typical for recursive data processing. To define the least fixpoint semantics of datalog$^\circ$, the semiring needs to be partially ordered. For this purpose, we introduce an algebraic structure called a ***Partially Ordered Pre-Semiring (POPS)***, which generalizes the more familiar naturally ordered semirings. This generalization is necessary for some applications. For example, the bill-of-material program (Example 4.2) is naturally expressed over the lifted reals, $\mathbb{R}_\perp$, which is a POPS that is not naturally ordered.

Like datalog, datalog$^\circ$ can be evaluated using the *naïve algorithm* by repeatedly applying all rules of the program until there is no more change. However, unlike datalog, a datalog$^\circ$ program may diverge. Our second contribution consists of a full characterization of the POPS that guarantee that every datalog$^\circ$ program terminates. More precisely, we show that termination is guaranteed iff the POPS enjoys a certain algebraic property called *stability* [33]. The result is based on an analysis of the fixpoint of a vector-valued polynomial function over a semiring, which is of independent interest. With some abuse, we will say in this article that a datalog$^\circ$ program *converges* if the naïve algorithm terminates in a finite number of steps; we do not consider "convergence in the limit," for example, in ω-continuous semirings [19, 38].

Finally, we describe how the *semi-naïve algorithm* can be extended to datalog$^\circ$, under certain restrictions on the POPS. This should be viewed as an illustration of the potential for applying advanced optimizations to datalog$^\circ$: In a companion paper [81], we introduced a simple, yet powerful optimization technique for datalog$^\circ$ and showed, among other things, that magic set rewriting can be obtained using several applications of that rule.

At its essence, a datalog° program consists of solving a fixpoint equation in a semiring, which is a problem that was studied in a variety of areas, such as automata theory, program analysis, and many others [12, 33, 41, 51, 52, 63, 72, 88]. The existence of the fixpoint is usually ensured by requiring the semiring to be ω-continuous. For example, Green et al. [38] studied the provenance of datalog on K-relations, while Esparza et al. [19] studied dataflow equations, in both cases requiring the semiring to be ω-continuous. This guarantees that the least fixpoint exists, even if the naïve algorithm diverges.

In addition to the (semi-)naïve method, which is a first-order method for solving fixpoint equations, there is a second-order method called the *Newton's method*, discovered relatively recently [19, 41] in the program analysis literature. It was shown that Newton's method requires a smaller number of iterations than the naïve algorithm.[1] In the particular case of a commutative and idempotent semiring, Newton's method always converges, while the naïve algorithm may diverge [19]. However, every iteration of Newton's method requires solving a least fixpoint solution to an inner-recursion, which is expensive. Here, we are seeing the exact same phenomenon when comparing gradient descent vs. Newton's method in continuous optimization: While Newton's method may converge in fewer steps, every step is more expensive and requires the materialization of a large intermediate result (the Hessian matrix). In continuous optimization, Newton's method is rarely used for large-scale problems for this reason [64]. For datalog°, it remains unclear whether Newton's method is more efficient in practice than the naïve algorithm. One experimental evaluation [69] has found that it is not. In contrast, the naïve algorithm, and its extension to the semi-naïve algorithm, is simple and implemented by all datalog systems today. In this article, we focus only on the convergence of the naïve algorithm and do not consider Newton's method.

A preliminary version of this article appeared in Reference [3]. The convergence theorem in stable semirings, Theorem 5.10, is new. The convergence semiring in p-stable semirings, Theorem 5.12, is improved and extended. We have also included many proofs omitted from Reference [3] and added several examples.

1.1 Overview of the Results

We present here a high-level overview of the main results in this article. We start by recalling the syntax of a (traditional) datalog program Π, which consists of a set of rules of the form:

$$R_0(\boldsymbol{X}_0) \text{ :- } R_1(\boldsymbol{X}_1) \wedge \cdots \wedge R_m(\boldsymbol{X}_m), \tag{1}$$

where $R_0, \dots, R_m$ are relation names (not necessarily distinct), and each $\boldsymbol{X}_i$ is a tuple of variables and/or constants. The atom $R_0(\boldsymbol{X}_0)$ is called the head, and the conjunction $R_1(\boldsymbol{X}_1) \wedge \cdots \wedge R_m(\boldsymbol{X}_m)$ is called the body. A relation name that occurs in the head of some rule in Π is called an ***intensional database predicate*****(IDB)**, otherwise it is called an ***extensional database predicate*****(EDB)**. The EDBs form the input database, which is denoted I, and the IDBs represent the output instance computed by Π, which we denote by J. The finite set of all constants occurring in I is called the *active domain* of I and denoted $\text{ADom}(I)$. The textbook example of a datalog program is one that computes the transitive closure of a graph defined by the edge relation $E(X, Y)$:

$$\begin{aligned} T(X, Y) &\text{ :- } E(X, Y) \\ T(X, Y) &\text{ :- } T(X, Z) \wedge E(Z, Y). \end{aligned}$$

Here, E is an EDB predicate, and T is an IDB predicate. Multiple rules with the same head are interpreted as a disjunction, and in this article, we prefer to combine them in a single rule with an

[1]The naïve algorithm is called Kleene iteration method in Reference [19].

explicit disjunction. In particular, the program above becomes:

$$T(X,Y) \text{ :- } E(X,Y) \vee \exists_Z (T(X,Z) \wedge E(Z,Y)). \tag{2}$$

The semantics of a datalog program is the least fixpoint of the function defined by its rules and can be computed by the *naïve evaluation algorithm*, as follows: Initialize all IDBs to the empty set, then repeatedly apply all rules, updating the state of the IDBs, until there is no more change to the IDBs. The algorithm always terminates in time polynomial in ADom(I).

We introduce datalog$^\circ$, a language that generalizes datalog to relations over a *partially ordered, commutative pre-semiring*, or POPS. A POPS, $\boldsymbol{P}$, is an algebraic structure with operations $\oplus, \otimes$ on a domain P, which is also a partially ordered set with an order relation $\sqsubseteq$ and a smallest element $\perp$ (formal definition is in Section 2). For example, the Boolean semiring $\mathbb{B}$ forms a POPS where $(\oplus, \otimes) = (\vee, \wedge)$, over domain $\{0, 1\}$, and the order relation is $0 \leq 1$. For another example, the *tropical semiring*, Trop, consists of the real numbers $\mathbb{R}$, the operations are min, $+$, and the order relation $x \sqsubseteq y$ is the reverse of the usual order $x \geq y$. If we restrict the tropical semiring to the non-negative reals and infinity, $\mathbb{R}_+ \cup \infty$, then we denote the POPS by Trop$^+$. For a fixed POPS $\boldsymbol{P}$, a $\boldsymbol{P}$-relation of arity k is a function that maps all k-tuples over some domain to values in $\boldsymbol{P}$; for example, a standard relation is a $\mathbb{B}$-relation, where $\mathbb{B}$ is the Boolean semiring, while a relation with duplicates (a bag) is an $\mathbb{N}$-relation. A datalog$^\circ$ program consists of a set of rules, e.g., like (2), where all relations are $\boldsymbol{P}$-relations, and the operators $\wedge, \vee, \exists_Z$ are replaced by $\otimes, \oplus, \bigoplus_Z$, and its semantics is defined as its least fixpoint, when it exists, or undefined otherwise. We will illustrate the utility of datalog$^\circ$ through multiple examples in this article and give here only a preview.

Example 1.1. The ***all-pairs shortest paths*** **(APSP)** problem is to compute the shortest path length $T(X,Y)$ between all pairs X, Y of vertices in the graph, given that $E(X,Y)$ = the length of the edge (X,Y) in the graph. We assume that both E and T are Trop$^+$-relations. The APSP problem can be expressed very compactly in datalog$^\circ$ as

$$T(X,Y) \text{ :- } E(X,Y) \oplus \bigoplus_Z T(X,Z) \otimes E(Z,Y), \tag{3}$$

where $(\oplus, \otimes) = (\min, +)$ are the "addition" and "multiplication" in Trop$^+$. If we instantiate these operations in (3), then the program becomes:

$$T(X,Y) \text{ :- } \min\left(E(X,Y), \min_Z (T(X,Z) + E(Z,Y))\right), \tag{4}$$

If we use a different POPS in (3), then the same datalog$^\circ$ program will express similar problems in exactly the same way. For example, if the relations E, T are $\mathbb{B}$-relations, then the program (3) becomes the transitive closure program (2); alternatively, if both E, T are Trop$_p^+$-relations, where Trop$_p^+$ is a semiring defined in Section 2, then the program computes the *top* $p+1$*-shortest-paths*.

The least fixpoint of datalog$^\circ$, if exists, can be computed by the naïve algorithm, quite similar to datalog: Initialize all IDBs to $\perp$, then repeatedly apply all rules of the datalog$^\circ$ program, obtaining an increasing chain of IDB instances, $J^{(0)} \sqsubseteq J^{(1)} \sqsubseteq J^{(2)} \sqsubseteq \cdots$. When $J^{(t)} = J^{(t+1)}$, then the algorithm stops and returns $J^{(t)}$; in that case, we say that the datalog$^\circ$ program converges in t steps, or we just say that it converges; otherwise, we say that it diverges. Traditional datalog always converges, but this is no longer the case for datalog$^\circ$ programs. There are five possibilities, depending on the choice of the POPS $\boldsymbol{P}$:

(i) For some datalog$^\circ$ programs, $\bigvee_t J^{(t)}$ is not the least fixpoint.
(ii) Every datalog$^\circ$ program has the least fixpoint $\bigvee_t J^{(t)}$, but may not necessarily converge.
(iii) Every datalog$^\circ$ program converges.

(iv) Every datalog$^\circ$ program converges in a number of steps that depends only on $|\text{ADom}(I)|$.
(v) Every datalog$^\circ$ program converges in a number of steps that is polynomial in $|\text{ADom}(I)|$.

In this article, we will only consider data-complexity [79], where the datalog$^\circ$ program is assumed to be fixed, and the input consists only of the EDB instance I.

We study algebraic properties of the POPS $\boldsymbol{P}$ that ensure that we are in one of the cases iii–v; we do not address cases i–ii in this article. We give next a necessary and sufficient condition for each of the cases iii and iv and give a sufficient condition for case v. For any POPS $\boldsymbol{P}$, the set $\boldsymbol{P} \oplus \bot \stackrel{\text{def}}{=} \{u \oplus \bot \mid u \in P\}$ is a semiring (Proposition 2.4); our characterization is based entirely on a certain property, called *stability*, of the semiring $\boldsymbol{P} \oplus \bot$, which we describe here.

Given a semiring $\boldsymbol{S}$ and $u \in S$, denote by $u^{(p)} := 1 \oplus u \oplus u^2 \oplus \cdots \oplus u^p$, where $u^i := u \otimes u \otimes \cdots \otimes u$ (i times). We say that u is *p-stable* if $u^{(p)} = u^{(p+1)}$; we say that the semiring $\boldsymbol{S}$ is *p-stable* if every element $u \in S$ is p-stable, and we say that $\boldsymbol{S}$ is *stable* if every element u is stable for some p that may depend on u. A datalog$^\circ$ program is *linear* if every rule has at most one IDB predicate in the body. We prove:

Theorem 1.2. *Given a POPS $\boldsymbol{P}$, the following hold:*

— *Every* datalog$^\circ$ *program converges iff the semiring $\boldsymbol{P} \oplus \bot$ is stable.*
— *Every program converges in a number of steps that depends only on $|ADom(I)|$ iff $\boldsymbol{P} \oplus \bot$ is p-stable for some p. More precisely, every* datalog$^\circ$ *program converges in $\sum_{i=1}^{N}(p+2)^i$ steps, where N is the number of ground tuples consisting of IDB predicates and constants from $ADom(I)$. Furthermore, if the program is linear, then it converges in $\sum_{i=1}^{N}(p+1)^i$ steps.*
— *If $\boldsymbol{P} \oplus \bot$ is 0-stable, then every* datalog$^\circ$ *program converges in N steps; in particular, the program runs in polynomial time in the size of the input database.*

In a nutshell, the theorem says that convergence of datalog$^\circ$ is intimately related to the notion of stability. The proof, provided in Section 5, consists of an analysis of the infinite powerseries resulting from unfolding the fixpoint definition; for the proof of the first item, we also use Parikh's theorem [65]. As mentioned earlier, most prior work on fixpoint equations assumes an ω-continuous semiring; when convergence is desired, one usually offers the **Ascending Chain Condition (ACC)** (see Section 3) as a sufficient condition for convergence. Our theorem implies that ACC is only a sufficient but not a necessary condition for convergence; for example, Trop^+ is 0-stable, and therefore every datalog$^\circ$ program converges on Trop^+, yet it does not satisfy the ACC condition. A somewhat related result is proven by Luttenberger and Schlund [59], who showed that, if 1 is p-stable, then Newton's method requires at most $N + \log\log p$ iterations. As mentioned earlier, each step of Newton's method requires the computation of another least fixpoint, hence, that result does not inform us on the convergence of the naïve algorithm.

Next, we introduce an extension of the semi-naïve evaluation algorithm to datalog$^\circ$. It is known that the naïve evaluation algorithm is inefficient in practice, because at each iteration it repeats all the computations that it has done at the previous iterations. Most datalog systems implement an improvement called the *semi-naïve* evaluation, which keeps track of the delta between the successive states of the IDBs and applies the datalog rules as a function of these deltas to obtain the new iteration's deltas. Semi-naïve evaluation is one of the major optimization techniques for evaluating datalog, however, it is defined only for programs that are monotone under set inclusion, and the systems that implement it enforce monotonicity, preventing the use of aggregation in recursion. Our second result consists of showing how to adapt the semi-naïve algorithm to datalog$^\circ$, under certain restrictions of the POPS $\boldsymbol{P}$, thus, enabling the semi-naïve algorithm to be applied to programs with aggregation in recursion.

Let us illustrate the semi-naïve algorithm on the APSP program in Example 1.1. For that, let us first review the standard semi-naïve algorithm for pure datalog on the transitive closure program (2). While the naïve algorithm starts with $T^{(0)}(X,Y) = \emptyset$ and repeats $T^{(t+1)}(X,Y) = E(X,Y) \vee \bigvee_Z T^{(t)}(X,Z) \wedge E(Z,Y)$ for $t = 0,1,2,\ldots$, the semi-naïve algorithm starts by initializing $T^{(1)}(X,Y) = \delta^{(0)}(X,Y) = E(X,Y)$, then performs the following steps for $t = 1,2,3,\ldots$:

$$\begin{aligned}\delta^{(t)}(X,Y) &= \left(\bigvee_z \delta^{(t-1)}(X,Z) \wedge E(Z,Y)\right) \setminus T^{(t)}(X,Y) \qquad (5)\\ T^{(t+1)}(X,Y) &= T^{(t)}(X,Y) \cup \delta^{(t)}(X,Y).\end{aligned}$$

Thus, the semi-naïve algorithm avoids re-discovering at iteration t tuples already discovered at iterations $1,2,\ldots,t-1$.

To apply the same principle to the APSP program (4), we need to define an appropriate "minus" $\ominus$ operator on the tropical semiring. This operator is:

$$v \ominus u = \begin{cases} v & \text{if } v < u \\ \infty & \text{if } v \geq u \end{cases}. \qquad (6)$$

The semi-naïve algorithm for the APSP problem in (3) is a mirror of that in Equation (5):

$$\begin{aligned}\delta^{(t)}(X,Y) &= (\min_Z \delta^{(t-1)}(X,Z) + E(Z,Y)) \ominus T^{(t)}(X,Y), \qquad (7)\\ T^{(t+1)}(X,Y) &= \min(T^{(t)}(X,Y), \delta^{(t)}(X,Y))\end{aligned}$$

where the difference operator $\ominus$ is defined in Equation (6). The reason why this algorithm is more efficient than the naïve algorithm is the fact that, in general, a database system needs to store only the tuples that are "present" in a relation, i.e., those whose value is $\neq \bot$. In our example, only those tuples $\delta^{(t)}(X,Y)$ whose value is $\neq \infty$ need to be stored. The $\ominus$ operator in Equation (7) checks if the new value $\min_Z \delta^{(t-1)}(X,Z) + E(Z,Y)$ is strictly less than the old value $T^{(t)}(X,Y)$; If not, then it returns ∞, signaling that the value of (X,Y) in both $\delta^{(t)}(X,Y)$ and $T^{(t)}(X,Y)$ does not need to be updated. As a consequence, only those tuples $T^{(t+1)}(X,Y)$ need to be processed at step t where the value has strictly decreased from the previous step.

Finally, the last topic we address in this article is a possible way of introducing negation in datalog$^\circ$. We will show that by interpreting datalog$^\circ$ over a particular POPS called THREE (Section 2.5.2) and by appropriately defining a function `not`, datalog$^\circ$ can express datalog queries with negation under Fitting's three-valued semantics [20]. The crux in Fitting's approach is to apply to the three-valued logic (with truth values $\{0,1,\bot\}$, where $\bot$ means "undefined") the *knowledge order* $\leq_k$ with $\bot \leq_k 0$ and $\bot \leq_k 1$. Then, the function `not` defined as $\texttt{not}(0) = 1$, $\texttt{not}(1) = 0$, and $\texttt{not}(\bot) = \bot$ is monotone w.r.t. $\leq_k$, and one can apply the usual least fixpoint semantics to datalog programs with negation (and, likewise, to datalog$^\circ$ programs with the function `not`). Hence, in cases when Fitting's 3-valued semantics coincides with the well-founded semantics, so does datalog$^\circ$ equipped with the function `not` when interpreted over the POPS THREE.

1.2 Article Organization

We define POPS in Section 2 and give several examples. In Section 3, we consider the least fixpoint of monotone functions over posets and prove an upper bound on the number of iterations needed to compute the least fixpoint. We define datalog$^\circ$ formally in Section 4 and give several examples. The convergence results described in Theorem 1.2 are stated formally and proven in Section 5. Section 6 presents a generalization of semi-naïve evaluation to datalog$^\circ$. We discuss how datalog$^\circ$ can express datalog queries with negation using 3-valued logic in Section 7. We conclude in Section 9.

2 PARTIALLY ORDERED PRE-SEMIRINGS

In this section, we review the basic algebraic notions of (pre-)semirings, P-relations, and sum-product queries. We also introduce an extension called **partially ordered pre-semiring (POPS)**.

2.1 (Pre-)Semirings and POPS

Definition 2.1 ((Pre-)semiring). A *pre-semiring* [33] 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 $\oplus$. When the *absorption rule* $x \otimes 0 = 0$ holds for all $x \in S$, we call $\boldsymbol{S}$ a *semiring.*[2] When $\otimes$ is commutative, then we say that the pre-semiring is *commutative.* In this article, we only consider commutative pre-semirings, and we will simply refer to them as pre-semirings.

In any (pre-)semiring $\boldsymbol{S}$, the relation $x \preceq_S y$ defined as $\exists z : x \oplus z = y$, is a *preorder*, which means that it is reflexive and transitive, but it is not anti-symmetric in general. When $\preceq_S$ is anti-symmetric, then it is a partial order and is called the *natural order* on $\boldsymbol{S}$; in that case, we say that $\boldsymbol{S}$ is *naturally ordered.*

Example 2.2. Some simple examples of naturally ordered semirings are the Booleans ($\mathbb{B} \overset{\text{def}}{=} \{0, 1\}, \vee, \wedge, 0, 1)$, the natural numbers $(\mathbb{N}, +, \times, 0, 1)$, and the real numbers $(\mathbb{R}, +, \times, 0, 1)$. We will refer to them simply as $\mathbb{B}, \mathbb{N}$, and $\mathbb{R}$. The natural order on $\mathbb{B}$ is $0 \preceq_{\mathbb{B}} 1$ (or false $\preceq_{\mathbb{B}}$ true); the natural order on $\mathbb{N}$ is the same as the familiar total order $\leq$ of numbers. $\mathbb{R}$ is not naturally ordered, because $x \preceq_{\mathbb{R}} y$ holds for every $x, y \in \mathbb{R}$. Another useful example is the *tropical semiring*$\text{Trop}^+ = (\mathbb{R}_+ \cup \{\infty\}, \min, +, \infty, 0)$, where the natural order $x \preceq y$ is the *reverse* order $x \geq y$ on $\mathbb{R}_+ \cup \{\infty\}$.

A key idea we introduce in this article is the decoupling of the partial order from the algebraic structure of the (pre-)semiring. The decoupling allows us to inject a partial order when the (pre-)semiring is not naturally ordered or when we need a *different* partial order than the natural order.

Definition 2.3 (POPS). A *partially ordered pre-semiring* (POPS), is a tuple $\boldsymbol{P} = (P, \oplus, \otimes, 0, 1, \sqsubseteq)$, where $(P, \oplus, \otimes, 0, 1)$ is a pre-semiring, $(P, \sqsubseteq)$ is a poset, and $\oplus, \otimes$ are *monotone*[3] operators under $\sqsubseteq$. In this article, we will assume that every poset $(P, \sqsubseteq)$ has a minimum element denoted by $\bot$.

A POPS satisfies the identities $\bot \oplus \bot = \bot$ and $\bot \otimes \bot = \bot$, because, by monotonicity and the fact that $(P, \oplus, 0)$ and $(P, \otimes, 1)$ are commutative monoids, we have $\bot \oplus \bot \sqsubseteq \bot \oplus 0 = \bot$, and $\bot \otimes \bot \sqsubseteq \bot \otimes 1 = \bot$. We say that the multiplicative operator $\otimes$ is *strict* if the identity $x \otimes \bot = \bot$ holds for every $x \in P$.

Throughout this article, we will assume that $\otimes$ is strict unless otherwise stated. One of the reasons for insisting on the strictness assumption is that we can "extract" from a POPS a semiring, called the *core semiring* of the POPS, as shown in the following simple proposition:

Proposition 2.4. *Given an arbitrary POPS $\boldsymbol{P} = (P, \oplus, \otimes, 0, 1, \sqsubseteq)$. Define the subset $P \oplus \bot \overset{def}{=} \{x \oplus \bot \mid x \in P\} \subseteq P$. Then, $(P \oplus \bot, \oplus, \otimes, 0 \oplus \bot, 1 \oplus \bot)$ is a semiring. We denote this semiring by $\boldsymbol{P} \oplus \bot$ and refer to it as the* core *semiring of $\boldsymbol{P}$.*

Proof. We use the fact that $\otimes$ is strict and check that $\boldsymbol{P} \oplus \bot$ is closed under $\oplus$ and $\otimes$: $(x \oplus \bot) \oplus (y \oplus \bot) = (x \oplus y) \oplus (\bot \oplus \bot) = (x \oplus y) \oplus \bot$, and similarly $(x \oplus \bot) \otimes (y \oplus \bot) = (x \otimes y) \oplus (x \otimes \bot) \oplus (y \otimes \bot) \oplus \bot = (x \otimes y) \oplus \bot \oplus \bot \oplus \bot = (x \otimes y) \oplus \bot \in \boldsymbol{P} \oplus \bot$. Finally, it suffices to observe that $\bot = 0 \oplus \bot$ is the identity for $\oplus$ and $1 \oplus \bot$ is the identity for $\otimes$. □

[2]Some references, e.g., Reference [47], define a semiring without absorption.
[3]Monotonicity means $x \sqsubseteq x'$ and $y \sqsubseteq y'$ imply $x \oplus y \sqsubseteq x' \oplus y'$ and $x \otimes y \sqsubseteq x' \otimes y'$.

Every naturally ordered semiring is a POPS, where $\bot = 0$ and $\otimes$ is strict, and its core is itself, $\boldsymbol{S} \oplus 0 = \boldsymbol{S}$. The converse does not hold: Some POPS are not naturally ordered; a simple example of a non-naturally ordered POPS is the set of *lifted reals*,[4] $\mathbb{R}_\bot \stackrel{\text{def}}{=} (\mathbb{R} \cup \{\bot\}, +, *, 0, 1, \sqsubseteq)$, where $x + \bot = x * \bot = \bot$ for all x, and $x \sqsubseteq y$ iff $x = \bot$ or $x = y$. Its core semiring is the trivial semiring $\mathbb{R}_\bot + \bot = \{\bot\}$ consisting of a single element. We will consider similarly the lifted natural numbers, $\mathbb{N}_\bot$.

2.2 Polynomials over POPS

Fix a POPS $\boldsymbol{P} = (P, \oplus, \otimes, 0, 1, \sqsubseteq)$. We are interested in analyzing behaviors of vector-valued multivariate functions on P defined by composing $\oplus$ and $\otimes$. These functions are multivariate polynomials. Writing polynomials in $\boldsymbol{P}$ using the symbols $\oplus, \otimes$ is cumbersome and difficult to parse. Consequently, we replace them with $+, \cdot$ when the underlying POPS $\boldsymbol{P}$ is clear from context; furthermore, we will also abbreviate a multiplication $a \cdot b$ with ab. As usual, a^k denotes the product of k copies of a, where $a^0 \stackrel{\text{def}}{=} 1$.

Let $x_1, \dots, x_N$ be N variables. A *monomial* (on $\boldsymbol{P}$) is an expression of the form:

$$m \stackrel{\text{def}}{=} c \cdot x_1^{k_1} \cdot \dots \cdot x_N^{k_N}, \tag{8}$$

where $c \in P$ is some constant. Its *degree* is $\deg(m) \stackrel{\text{def}}{=} k_1 + \dots + k_N$. A (multivariate) *polynomial* is a sum:

$$f(x_1, \dots, x_N) \stackrel{\text{def}}{=} m_1 + m_2 + \dots + m_q, \tag{9}$$

where each m_i is a monomial. The polynomial f defines a function $P^N \to P$ in the obvious way, and, with some abuse, we will denote by f both the polynomial and the function it defines. Notice that f is monotone in each of its arguments.

A *vector-valued polynomial function* on $\boldsymbol{P}$ is a function $\mathbf{f} : P^N \to P^M$ whose component functions are polynomials. In particular, the vector-valued polynomial function is a *tuple of polynomials* $\mathbf{f} = (f_1, \dots, f_M)$ where each f_i is a polynomial in variables $x_1, \dots, x_N$.

We note a subtlety when dealing with POPS: When the POPS is not a semiring, then we cannot "remove" monomials by setting their coefficient $c = 0$, because 0 is not absorbing. Instead, we must ensure that they are not included in the polynomial (9). For example, consider the POPS of the lifted reals, $\mathbf{R}_\bot$ and the linear polynomial $f(x) = ax + b$. If we set $a = 0$, then we do not obtain the constant function $g(x) = b$, because $f(\bot) = a\bot + b = \bot + b = \bot \neq g(\bot) = b$. We just have to be careful to not include monomials we do not want.

2.3 *P*-Relations

Fix a relational vocabulary, $\sigma = \{R_1, \dots, R_m\}$, where each R_i is a relation name, with an associated arity. Let D be an infinite domain of constants, for example, the set of all strings over a fixed alphabet or the set of natural numbers. Recall that an instance of the relation R_i is a finite subset of $D^{\text{arity}(R_i)}$, or equivalently, a mapping $D^{\text{arity}(R_i)} \to \mathbb{B}$ assigning 1 to all tuples present in the relation. Following Reference [38], we generalize this abstraction from $\mathbb{B}$ to an arbitrary POPS $\boldsymbol{P}$.

Given a relation name $R_i \in \sigma$, a *ground atom* of R_i is an expression of the form $R_i(\mathbf{u})$, where $\mathbf{u} \in D^{\text{arity}(R_i)}$. Let $\text{GA}(R_i, D)$ denote the set of all ground atoms of R_i, and $\text{GA}(\sigma, D) \stackrel{\text{def}}{=} \bigcup_i \text{GA}(R_i, D)$ denotes the set of all ground atoms over the entire vocabulary σ. The set $\text{GA}(\sigma, D)$ is the familiar Herbrand base in logic programming. Note that each ground atom is prefixed by a relation name.

Let $\boldsymbol{P}$ be a POPS. A *$\boldsymbol{P}$-instance for σ* is a function $I : \text{GA}(\sigma, D) \to \boldsymbol{P}$ with *finite support*, where the support is defined as the set of ground atoms that are mapped to elements *other than* $\bot$. For

[4]The term "lifted" comes from the programming languages community where adding bottom to a set "lifts" the set.

example, if $\boldsymbol{P}$ is a naturally ordered semiring, then the support of the function I is the set of ground atoms assigned to a non-zero value. The *active domain* of the instance I, denoted by $\mathrm{ADom}(I)$, is the finite set $\mathrm{ADom}(I) \subseteq D$ of all constants that occur in the support of I. We denote by $\mathsf{Inst}(\sigma, D, \boldsymbol{P})$ the set of $\boldsymbol{P}$-instances over the domain D. When σ consists of a single relation name, then we call I a *$\boldsymbol{P}$-relation*.

An equivalent way to define a $\boldsymbol{P}$-instance is as a function $I : \mathrm{GA}(\sigma, D_0) \to \boldsymbol{P}$, for some finite subset $D_0 \subseteq D$; by convention, this function is extended to the entire set $\mathrm{GA}(\sigma, D)$ by setting $I(a) := \bot$ for all $a \in \mathrm{GA}(\sigma, D) \setminus \mathrm{GA}(\sigma, D_0)$. The set $\mathsf{Inst}(\sigma, D_0, \boldsymbol{P})$ is isomorphic to P^N, where $N = |\mathrm{GA}(\sigma, D_0)|$, and, throughout this article, we will identify a $\boldsymbol{P}$-instance with a tuple in P^N.

Thus, a $\boldsymbol{P}$-instance involves two domains: D, which is called the *key space*, and the POPS $\boldsymbol{P}$, which is called the *value space*. For some simple illustrations, a $\mathbb{B}$-relation is a standard relation where every ground tuple is either true or false, while an $\mathbb{R}$-relation is a sparse tensor.

2.4 (Sum-)Sum-product Queries on POPS

In the Boolean world, conjunctive queries and unions of conjunctive queries are building-block queries. Analogously, in the POPS world, we introduce the concepts of *sum-product queries* and *sum-sum-product queries*. In the most simple setting, these queries have been studied in other communities (especially AI and machine learning, as reviewed below). In our setting, we need to introduce one extra feature called "conditional" to cope with the fact that 0 is not absorptive.

Fix two disjoint vocabularies, $\sigma, \sigma_{\mathbb{B}}$; the relation names in σ will be interpreted over a POPS $\boldsymbol{P}$, while those in $\sigma_{\mathbb{B}}$ will be interpreted over the Booleans. Let D be a domain and $V = \{X_1, \ldots, X_p\}$ a set of "key variables" whose values are over the key space D. They should not be confused with variables used in polynomials, which are interpreted over the POPS $\boldsymbol{P}$; we refer to them as "value variables" to contrast them with the key variables. We use upper case for key variables and lower case for value variables. A *σ-atom* is an expression of the form $R_i(\boldsymbol{X})$, where $R_i \in \sigma$ and $\boldsymbol{X} \in (V \cup D)^{\mathrm{arity}(R_i)}$.

Definition 2.5. A (conditional) *sum-product query*, or *sum-product rule,* is an expression of the form

$$T(X_1, \ldots, X_k) \text{ :- } \bigoplus_{X_{k+1}, \ldots, X_p} \{R_1(\boldsymbol{X}_1) \otimes \cdots \otimes R_m(\boldsymbol{X}_m) \mid \Phi(V)\}, \tag{10}$$

where T is a new relation name of arity k, each $R_j(\boldsymbol{X}_j)$ is a σ-atom, and Φ is a **first-order (FO)** formula over $\sigma_{\mathbb{B}}$, whose free variables are in $V = \{X_1, \ldots, X_p\}$. The LHS of :- is called the *head*, and the RHS the *body* of the rule. The variables $X_1, \ldots, X_k$ are called *free variables* of the query (also called *head variables*), and $X_{k+1}, \ldots, X_p$ are called *bound variables.*

Without the conditional term Φ, the problem of computing efficiently 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-product computation over the value-space of Booleans (set semantics) or natural numbers (bag semantics). The **functional aggregate queries (FAQ)** framework [4] extends the formulation to queries over multiple semirings. In AI, this problem was studied by Shenoy and Schafer [73], Dechter [18], Kohlas and Wilson [47], and others. Surveys and more examples can be found in References [5, 46]. These methods use a sparse representation of the $\boldsymbol{P}$-relations, consisting of a collection of the tuples in their support.

The use of a conditional Φ in the sum-product is non-standard, but it is necessary for sum-product expressions over a POPS that is not a semiring, as we illustrate next.

Example 2.6. Let $E(X, Y)$ be a $\mathbb{B}$-relation (i.e., a standard relation), representing a graph. The following sum-product expression over $\mathbb{B}$ computes all pairs of nodes connected by a path of length 2:

$$T(X,Z) \text{ :- } \exists_Y \left(E(X,Y) \wedge E(Y,Z)\right).$$

This is a standard conjunctive query [2] (where the semantics of quantification over Y is explicitly written). Here, $\sigma = \{E\}$, and $\sigma_{\mathbb{B}} = \emptyset$: We do not need the conditional term Φ yet.

For the second example, consider the same graph given by $E(X,Y)$, and let $C(X)$ be an $\mathbb{R}_\perp$-relation associating to each node X a real number representing a cost, or $\perp$ if the cost is unknown; now $\sigma = \{C\}, \sigma_{\mathbb{B}} = \{E\}$. The following sum-product expression computes the total costs of all neighbors of X:

$$T(X) \text{ :- } \sum_Y \{C(Y) \mid E(X,Y)\}. \tag{11}$$

Usually, conditionals are avoided by using an indicator function $1_{E(X,Y)}$, which is defined to be 1 when $E(X,Y)$ is true and 0 otherwise, and writing the rule as $T(X) \text{ :- } \sum_Y \left(1_{E(X,Y)} \cdot C(Y)\right)$. But this does not work in $\mathbb{R}_\perp$, because, when Y is mapped to a non-neighboring node that so happens to have an unknown cost (while all neighbors' costs are known), we have $C(Y) = \perp$. In this case, $1_{E(X,Y)} \cdot C(Y) = 0 \cdot \perp = \perp$, instead of 0. Since $x + \perp = \perp$ in $\mathbb{R}_\perp$, the result is also $\perp$. One may ask whether we can re-define the POPS $\mathbb{R}_\perp$ so $\perp \cdot 0 = 0$, but we show in Lemma 2.8 that this is not possible. The explicit conditional in (11) allows us to restrict the range of Y only to the neighbors of X.

We now formally define the semantics of (conditional) sum-product queries. Due to the subtlety with POPS, we need to consider an alternative approach to evaluating the results of sum-product queries: First compute the *provenance polynomials* of the query (10) to obtain the component polynomials of a vector-valued function, then evaluate these polynomials. The provenance polynomials, or simply provenance, are also called lineage or groundings in the literature [38].

Given an input instance $I_{\mathbb{B}} \in \mathsf{Inst}(\sigma_{\mathbb{B}}, D, \mathbb{B})$, $I \in \mathsf{Inst}(\sigma, D, \boldsymbol{P})$, and let $D_0 \subseteq D$ be the finite set consisting of their active domains and all constants occurring in the sum-product expression (10). Let $N \stackrel{\text{def}}{=} |\mathrm{GA}(\sigma, D_0)|$ and $M \stackrel{\text{def}}{=} |\mathrm{GA}(T, D_0)| = |D_0|^k$ be the number of input ground atoms and output ground atoms, respectively.

To each of the N input atoms $\mathrm{GA}(\sigma, D_0)$, we associate a unique POPS variable $x_1, \dots, x_N$. (Recall that we use upper case for key variables and lower case for value variables.) Abusing notation, we also write $x_{R(\mathbf{u})}$ to mean the variable associated to the ground atom $R(\mathbf{u})$. A *valuation* is a function $\theta : V \longrightarrow D_0$. When applied to the body of the rule (10), the valuation θ defines the following monomial:

$$\theta(\text{body}) \stackrel{\text{def}}{=} x_{R_1(\theta(\boldsymbol{X}_1))} \cdot x_{R_2(\theta(\boldsymbol{X}_2))} \cdots x_{R_m(\theta(\boldsymbol{X}_m))}. \tag{12}$$

The *provenance polynomial* [38] of the output tuple $T(\mathbf{a}) \in \mathrm{GA}(T, D_0)$ is the following:

$$f_{T(\mathbf{a})}(x_1, \dots, x_N) \stackrel{\text{def}}{=} \sum_{\substack{\theta: V \to D_0: \\ \theta(X_1, \dots, X_k) = \mathbf{a}, \\ I_{\mathbb{B}} \models \Phi[\theta]}} \theta(\text{body}). \tag{13}$$

In other words, we consider only valuations θ that map the head variables to the tuple $\mathbf{a}$ and satisfy the FO sentence Φ. There are M provenance polynomials, one for each tuple in $\mathrm{GA}(T, D_0)$, and they define an M-tuple of polynomials in N variables, $\mathbf{f}$, which in turn defines a function $\mathbf{f} : \boldsymbol{P}^N \longrightarrow \boldsymbol{P}^M$. The semantics of the query (10) is defined as the value of this polynomial on the input instance $I \in \mathsf{Inst}(\sigma, D_0, \boldsymbol{P})$, when viewed as a tuple $I \in \boldsymbol{P}^N$.

Note that, once we have constructed the provenance polynomial, we no longer need to deal with the conditional Φ, because the grounded version does not have Φ anymore. In most of the rest of the article, we will study properties of vector-valued functions whose components are these provenance polynomials.

We notice that, as defined, our semantics depends on the choice of the domain D_0: If we used a larger finite domain $D'_0 \supseteq D_0$, then the provenance polynomials will include additional spurious monomials, corresponding to the spurious grounded tuples in D'_0. Traditionally, these spurious monomials are harmless, because their value is 0. However, in our setting, their value is $\perp$, and they may change the result. This is precisely the role of the conditional Φ in (10): to control the range of the variables and ensure that the semantics is domain independent. All examples in this article are written such that they are domain-independent.

Finally, (conditional) sum-sum-product queries are defined in the natural way:

Definition 2.7. A (conditional) *sum-sum-product query* or *sum-sum-product rule* has the form:

$$T(X_1, \ldots, X_k) \text{ :- } E_1 \oplus \cdots \oplus E_q, \tag{14}$$

where $E_1, E_2, \ldots, E_q$ are the bodies of sum-product expressions (10), each with the same free variables $X_1, \ldots, X_k$.

The provenance polynomials of a sum-sum-product query are defined as the sum of the provenance polynomials of the expressions $E_1, \ldots, E_q$.

2.5 Properties and Examples of POPS

We end this section by presenting several properties of POPS and illustrating them with a few examples.

2.5.1 Extending Pre-semirings to POPS. If $\boldsymbol{S}$ is a pre-semiring, then we say that a POPS $\boldsymbol{P}$ *extends* $\boldsymbol{S}$ if $S \subseteq P$ (S and P are their domains), and the operations $\oplus, \otimes, 0, 1$ in $\boldsymbol{S}$ are the same as those in $\boldsymbol{P}$. We describe three procedures to extend a pre-semiring $\boldsymbol{S}$ to a POPS $\boldsymbol{P}$, inspired by abstract interpretations in programming languages [12].

Representing Undefined. The *lifted POPS* is $\boldsymbol{S}_\perp = (S \cup \{\perp\}, \oplus, \otimes, 0, 1, \sqsubseteq)$, where $x \sqsubseteq y$ iff $x = \perp$ or $x = y$, and the operations $\oplus, \otimes$ are extended to $\perp$ by setting $x \oplus \perp = x \otimes \perp = \perp$. Notice that $\boldsymbol{S}_\perp$ is not a semiring, because 0 is not absorbing: $0 \otimes \perp \neq 0$. Its core semiring is the trivial semiring $S_\perp \oplus \perp = \{\perp\}$. Here, $\perp$ represents *undefined*.

Representing Contradiction. The *completed POPS* is $\boldsymbol{S}_\perp^\top = (S \cup \{\perp, \top\}, \oplus, \otimes, 0, 1, \sqsubseteq)$, where $x \sqsubseteq y$ iff $x = \perp$, $x = y$, or $y = \top$ and the operations $\oplus, \otimes$ are extended to $\perp, \top$ as follows: $x \oplus \perp = x \otimes \perp = \perp$ for all x (including $x = \top$), and $x \oplus \top = x \otimes \top = \top$ for all $x \neq \perp$. As before, its core semiring is the trivial semiring $S_\perp^\top \oplus \perp = \{\perp\}$. Here, $\perp, \top$ represent undefined and contradiction, respectively. Intuitively: $\perp$ is the empty set $\emptyset$, each element $x \in \boldsymbol{S}$ is a singleton set consisting of one value, and $\top$ is the entire set S.

Representing Incomplete Values. More generally, define $\mathcal{P}(\boldsymbol{S}) = (\mathcal{P}(S), \oplus, \otimes, 0, 1, \subseteq)$. It consists of all subsets of $\boldsymbol{S}$, ordered by set inclusion, where the operations $\oplus, \otimes$ are extended to sets, e.g., $A \oplus B = \{x \oplus y \mid x \in A, y \in B\}$. Its core semiring is itself, $\mathcal{P}(\boldsymbol{S}) \oplus \{0\} = \mathcal{P}(\boldsymbol{S})$. Here, $\perp = \emptyset$ represents undefined, $\top = S$ represents contradiction, and, more generally, every set represents some degree of incompleteness.

A lifted POPS $\boldsymbol{S}_\perp$ is never a semiring, because $\perp \otimes 0 = \perp$, and the reader may ask whether there exists an alternative way to extend it to a POPS that is also a semiring, i.e., $0 \otimes x = 0$. For example, we can define $\mathbb{N} \cup \{\perp\}$ as a semiring by setting $x + \perp = \perp$ for all x, $0 \cdot \perp = 0$ and $x \cdot \perp = \perp$ for $x > 0$: One can check that the semiring laws hold. However, this is not possible in general. We prove:

LEMMA 2.8. *If S is any POPS extension of $(\mathbb{R}, +, \cdot, 0, 1)$, then S is not a semiring, i.e., it fails the absorption law $0 \cdot x = 0$.*

PROOF. Let $\boldsymbol{S} = (S, +, \cdot, 0, 1, \sqsubseteq)$ be any POPS that is an extension of $\mathbb{R}$. In particular, $\mathbb{R} \subseteq S$ and S has a minimal element $\perp$. Since 0, 1 are additive and multiplicative identities, we have:

$$\perp + 0 = \perp \qquad\qquad \perp \cdot 1 = \perp.$$

We claim that the following more general identities hold:

$$\forall x \in \mathbb{R} : \ \perp + x = \perp \qquad\qquad \forall x \in \mathbb{R} \setminus \{0\} : \perp \cdot x = \perp.$$

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

Assuming $\boldsymbol{S}$ is a semiring, it satisfies the absorption law: $\perp \cdot 0 = 0$. We prove now that $0 = \perp$. Choose any $x \in \mathbb{R} \setminus \{0\}$, and derive:

$$\perp = \perp + \perp = (\perp \cdot x) + (\perp \cdot (-x)) = \perp \cdot (x + (-x)) = \perp \cdot 0 = 0.$$

The middle identity follows from distributivity. From $0 = \perp$, we conclude that 0 is the smallest element in $\boldsymbol{S}$. Then, for every $x \in \mathbb{R}$, we have $x = x + 0 \sqsubseteq x + (-x) = 0$, which implies $x = 0, \forall x \in \mathbb{R}$, which is a contradiction. Thus, $\boldsymbol{S}$ is not a semiring. □

2.5.2 *The POPS THREE.* Consider the following POPS: $\mathsf{THREE} \stackrel{\text{def}}{=} (\{\perp, 0, 1\}, \vee, \wedge, 0, 1, \leq_k)$, where:

- $\vee, \wedge$ have the semantics of 3-valued logic [20]. More precisely, define the *truth ordering* $0 \leq_t \perp \leq_t 1$ and set $x \vee y \stackrel{\text{def}}{=} \max_t(x, y)$, $x \wedge y \stackrel{\text{def}}{=} \min_t(x, y)$. We note that this is precisely Kleene's three-valued logic [22].
- $\leq_k$ is the *knowledge order*, defined as $\perp <_k 0$ and $\perp <_k 1$.

THREE is not the same as the lifted Booleans, $\mathbb{B}_\perp$, because in the latter $0 \wedge \perp = \perp$, while in THREE we have $0 \wedge \perp = 0$. Its core semiring is $\mathsf{THREE} \vee \perp = \{\perp, 1\}$, and is isomorphic to $\mathbb{B}$. We will return to this POPS in Section 7.

2.5.3 *Stable Semirings.* We illustrate here two examples of semirings that are *stable*, a property that we define formally in Section 5. Both examples are adapted from Reference [33, Example 7.1.4] and Reference [33, Chapter 8, Section 1.3.2], respectively. If A is a set and $p \geq 0$ a natural number, then we denote by $\mathcal{P}_p(A)$ the set of subsets of A of size p, and by $\mathcal{B}_p(A)$ the set of bags of A of size p. We also define

$$\mathcal{P}_{\mathtt{fin}}(A) \stackrel{\text{def}}{=} \bigcup_{p \geq 0} \mathcal{P}_p(A) \qquad\qquad \mathcal{B}_{\mathtt{fin}}(A) \stackrel{\text{def}}{=} \bigcup_{p \geq 0} \mathcal{B}_p(A).$$

We denote bags as in $\{\{a, a, a, b, c, c\}\}$. Given $\mathbf{x}, \mathbf{y} \in \mathcal{P}_{\mathtt{fin}}(\mathbb{R}_+ \cup \infty)$, we denote by:

$$\mathbf{x} \cup \mathbf{y} \stackrel{\text{def}}{=} \text{set union of } \mathbf{x}, \mathbf{y} \qquad\qquad \mathbf{x} + \mathbf{y} \stackrel{\text{def}}{=} \{u + v \mid u \in \mathbf{x}, v \in \mathbf{y}\}.$$

Similarly, given $\mathbf{x}, \mathbf{y} \in \mathcal{B}_{fin}(\mathbb{R}_+ \cup \infty)$, we denote by:

$$\mathbf{x} \uplus \mathbf{y} \stackrel{\text{def}}{=} \text{bag union of } \mathbf{x}, \mathbf{y} \qquad\qquad \mathbf{x} + \mathbf{y} \stackrel{\text{def}}{=} \{\{u + v \mid u \in \mathbf{x}, v \in \mathbf{y}\}\}.$$

Example 2.9. For any bag $\mathbf{x} = \{\{x_0, x_1, \dots, x_n\}\}$, where $x_0 \leq x_1 \leq \cdots \leq x_n$, and any $p \geq 0$, define:

$$\min_p(\mathbf{x}) \overset{\text{def}}{=} \{\{x_0, x_1, \dots, x_{\min(p,n)}\}\}.$$

In other words, $\min_p$ returns the smallest $p+1$ elements of the bag $\mathbf{x}$. Then, for any $p \geq 0$, the following is a semiring:

$$\mathsf{Trop}_p^+ \overset{\text{def}}{=} (\mathcal{B}_{p+1}(\mathbb{R}_+ \cup \{\infty\}), \oplus_p, \otimes_p, \mathbf{0}_p, \mathbf{1}_p),$$

where:

$$\mathbf{x} \oplus_p \mathbf{y} \overset{\text{def}}{=} \min_p(\mathbf{x} \uplus \mathbf{y}) \qquad \mathbf{0}_p \overset{\text{def}}{=} \{\{\infty, \infty, \dots, \infty\}\}$$

$$\mathbf{x} \otimes_p \mathbf{y} \overset{\text{def}}{=} \min_p(\mathbf{x} + \mathbf{y}) \qquad \mathbf{1}_p \overset{\text{def}}{=} \{\{0, \infty, \dots, \infty\}\}.$$

For example, if $p = 2$, then $\{\{3,7,9\}\} \oplus_2 \{\{3,7,7\}\} = \{\{3,3,7\}\}$ and $\{\{3,7,9\}\} \otimes_2 \{\{3,7,7\}\} = \{\{6,10,10\}\}$. The following identities are easily checked, for any two finite bags $\mathbf{x}, \mathbf{y}$:

$$\min_p(\min_p(\mathbf{x}) \uplus \min_p(\mathbf{y})) = \min_p(\mathbf{x} \uplus \mathbf{y}) \qquad \min_p(\min_p(\mathbf{x}) + \min_p(\mathbf{y})) = \min_p(\mathbf{x} + \mathbf{y}). \tag{15}$$

This implies that an expression in the semiring Trop_p^+ can be computed as follows: First, convert $\oplus, \otimes$ to $\uplus, +$, respectively, compute the resulting bag, then apply $\min_p$ only once, on the final result. Trop_p^+ is naturally ordered (see Proposition 5.3) and therefore its core semiring is itself, $\mathsf{Trop}_p^+ \oplus_p \mathbf{0}_p = \mathsf{Trop}_p^+$. When $p = 0$, then $\mathsf{Trop}_p^+ = \mathsf{Trop}^+$, which we introduced in Example 1.1.

Example 2.10. Fix a real number $\eta \geq 0$, and denote by $\mathcal{P}_{\leq\eta}(\mathbb{R}_+ \cup \{\infty\})$ the set of nonempty, finite sets $\mathbf{x} = \{x_0, x_1, \dots, x_p\}$ where $\min(\mathbf{x}) \leq \max(\mathbf{x}) \leq \min(\mathbf{x}) + \eta$. Given any finite set $\mathbf{x} \in \mathcal{P}_{\mathtt{fin}}(\mathbb{R}_+ \cup \{\infty\})$, we define

$$\min_{\leq\eta}(\mathbf{x}) \overset{\text{def}}{=} \{u \mid u \in \mathbf{x}, u \leq \min(\mathbf{x}) + \eta\}.$$

In other words, $\min_{\leq\eta}$ retains from the set $\mathbf{x}$ only the elements at distance $\leq \eta$ from its minimum. The following is a semiring:

$$\mathsf{Trop}_{\leq\eta}^+ \overset{\text{def}}{=} (\mathcal{P}_{\leq\eta}(\mathbb{R}_+ \cup \{\infty\}), \oplus_{\leq\eta}, \otimes_{\leq\eta}, \mathbf{0}_{\leq\eta}, \mathbf{1}_{\leq\eta}),$$

where:

$$\mathbf{x} \oplus_{\leq\eta} \mathbf{y} \overset{\text{def}}{=} \min_{\leq\eta}(\mathbf{x} \cup \mathbf{y}) \qquad \mathbf{0}_{\leq\eta} \overset{\text{def}}{=} \{\infty\}$$

$$\mathbf{x} \otimes_{\leq\eta} \mathbf{y} \overset{\text{def}}{=} \min_{\leq\eta}(\mathbf{x} + \mathbf{y}) \qquad \mathbf{1}_{\leq\eta} \overset{\text{def}}{=} \{0\}.$$

For example, if $\eta = 6.5$, then: $\{3,7\} \oplus_{\leq\eta} \{5,9,10\} = \{3,5,7,9\}$ and $\{1,6\} \otimes_{\leq\eta} \{1,2,3\} = \{2,3,4,7,8\}$. The following identities are easily checked, for any two finite sets $\mathbf{x}, \mathbf{y}$:

$$\min_{\leq\eta}(\min_{\leq\eta}(\mathbf{x}) \cup \min_{\leq\eta}(\mathbf{y})) = \min_{\leq\eta}(\mathbf{x} \cup \mathbf{y}) \quad \min_{\leq\eta}(\min_{\leq\eta}(\mathbf{x}) + \min_{\leq\eta}(\mathbf{y})) = \min_{\leq\eta}(\mathbf{x} + \mathbf{y}). \tag{16}$$

It follows that expressions in $\mathsf{Trop}_{\leq\eta}^+$ can be computed as follows: First, convert $\oplus, \otimes$ to $\cup, +$, respectively, compute the resulting set, and apply the $\min_{\leq\eta}$ operator only once, on the final result. $\mathsf{Trop}_{\leq\eta}^+$ is naturally ordered (see Proposition 5.4) and therefore its core semiring is itself, $\mathsf{Trop}_{\leq\eta}^+ \oplus_{\leq\eta} \mathbf{0}_{\leq\eta} = \mathsf{Trop}_{\leq\eta}^+$. Notice that, when $\eta = 0$, then we recover again $\mathsf{Trop}_{\leq\eta}^+ = \mathsf{Trop}^+$.

The reader may wonder why Trop_p^+ is defined to consist of bags of $p+1$ numbers, while $\mathsf{Trop}_{\leq\eta}^+$ is defined on sets. The main reason is for consistency with Reference [33]. We could have defined either semirings on either sets or bags, and both identities (15) and (16) continue to hold, which is sufficient to prove the semiring identities. However, the *stability* property that we define and prove later (Proposition 5.4) holds for $\mathsf{Trop}_{\leq\eta}^+$ only if it is defined over sets; in contrast, Trop_p^+ is stable for either sets or bags (Proposition 5.3).

2.5.4 Nontrivial Core Semiring. In all our examples so far the core semiring $\boldsymbol{P} \oplus \bot$ is either $\{\bot\}$ or $\boldsymbol{P}$. We show next that the core semiring may be non-trivial. If $\boldsymbol{P}_1, \boldsymbol{P}_2$ are two POPS, then their Cartesian product $\boldsymbol{P}_1 \times \boldsymbol{P}_2$ is also a POPS: Operations are defined element-wise, e.g., $(x_1, x_2) \oplus (y_1, y_2) \stackrel{\text{def}}{=} (x_1 \oplus_1 y_1, x_2 \oplus_2 y_2)$, and so on, the order is defined component-wise, and the smallest element is $(\bot_1, \bot_2)$.

Example 2.11. Consider the following two POPS:

- A naturally ordered semiring $\boldsymbol{S} = (S, \oplus_S, \otimes_S, 0_S, 1_S, \sqsubseteq_S)$. Its core semiring is itself $\boldsymbol{S} \oplus_S 0_S = \boldsymbol{S}$.
- Any POPS $\boldsymbol{P}$ where addition is strict: $x \oplus_P \bot = \bot$. (For example, any lifted semiring.) Its core semiring is $\boldsymbol{P} \oplus_P \bot_P = \{\bot_P\}$.

Consider the Cartesian product $\boldsymbol{S} \times \boldsymbol{P}$. The smallest element is $(0_S, \bot_P)$, and the core semiring is $(\boldsymbol{S} \times \boldsymbol{P}) \oplus (0_S, \bot_P) = \boldsymbol{S} \times \{\bot_P\}$, which is a non-trivial subset of $\boldsymbol{S} \times \boldsymbol{P}$.

3 LEAST FIXPOINT

We review here the definition of a least fixpoint and prove some results needed to characterize the convergence of datalog$^\circ$ programs. Fix a ***partially ordered set*****(poset)**, $\mathbf{L} = (L, \sqsubseteq)$. As mentioned in Section 2, in this article, we will assume that each poset has a minimum element $\bot$ unless explicitly specified otherwise. 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. We assume the usual definition of a monotone function f between two posets, namely, $f(x) \sqsubseteq f(y)$ whenever $x \sqsubseteq y$. The Cartesian product of two posets $\mathbf{L}_1 = (L_1, \sqsubseteq_1)$ and $\mathbf{L}_2 = (L_2, \sqsubseteq_2)$, is also the standard component-wise ordering: $\mathbf{L}_1 \times \mathbf{L}_2 \stackrel{\text{def}}{=} (L_1 \times L_2, \sqsubseteq)$, where $(x_1, x_2) \sqsubseteq (y_1, y_2)$ if $x_1 \sqsubseteq_1 y_1$ and $x_2 \sqsubseteq_2 y_2$. An *ω-chain* in a poset $\mathbf{L}$ is a sequence $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \cdots$, or, equivalently, it is a monotone function $\mathbb{N} \to \mathbf{L}$. We say that the chain is finite if there exists n_0 such that $x_{n_0} = x_{n_0+1} = x_{n_0+2} = \cdots$, or, equivalently, if $x_{n_0} = \bigvee x_n$.

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

$$f^{(0)}(\bot) \stackrel{\text{def}}{=} \bot \qquad\qquad f^{(n+1)}(\bot) \stackrel{\text{def}}{=} f(f^{(n)}(\bot)). \tag{17}$$

If x is any fixpoint of f, then $f^{(n)}(\bot) \sqsubseteq x$; this follows immediately by induction on n. To ensure that the least fixpoint exists, several authors require the semiring to be ω-complete and f to be ω-continuous. In that case, the least upper bound $\bigvee_n f^{(n)}(\bot)$ always exists and is equal to $\mathsf{lfp}(f)$, due to Kleene's theorem; see Reference [17]. This condition was used extensively in the formal language literature [12, 51] and also by Green et al. [38]. We do not use this condition in this article and will not define it formally.

Instead, we are interested in conditions that ensure that the sequence (17) reaches a fixpoint after a *finite* number of steps, which justifies the following definition:

Definition 3.1. A monotone function f on $\mathbf{L}$ (i.e., $f : \mathbf{L} \to \mathbf{L}$) 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 f is p-stable, then $\mathsf{lfp}(f)$ exists and is equal to $f^{(p)}(\bot)$. Indeed, $f^{(p)}(\bot)$ is a fixpoint of f by definition, and, as mentioned earlier, it is below any fixpoint x of f. In this case, we will also say that the sequence (17) *converges*.

A sufficient condition for convergence often found in the literature is the ***Ascending Chain Condition*****(ACC)**; see, e.g., References [19, 63]: The poset $\mathbf{L}$ satisfies ACC if it has *no infinite*

ω-chains, meaning that every strictly increasing chain $x_0 \sqsubset x_1 \sqsubset x_2 \sqsubset \cdots$ must be finite. If $\mathbf{L}$ satisfies ACC, then every function $\mathbf{f}$ on $\mathbf{L}$ is stable and thus has a least fixpoint. One can also check that, if $\mathbf{L}_1, \ldots, \mathbf{L}_n$ satisfy ACC, then so does $\mathbf{L} \stackrel{\text{def}}{=} \mathbf{L}_1 \times \cdots \times \mathbf{L}_n$. However, as we will see shortly in Section 5, when f is a polynomial, then the ACC is only a sufficient, but not necessary, condition for stability.

In this article, we study the stability of functions on a product of posets. To this end, we start by considering two posets, $\mathbf{L}_1$ and $\mathbf{L}_2$, with minimum elements $\bot_1$ and $\bot_2$, respectively, and two monotone functions:

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

Let h be the vector-valued function with components f and g, i.e., $h \stackrel{\text{def}}{=} (f, g) : \mathbf{L}_1 \times \mathbf{L}_2 \to \mathbf{L}_1 \times \mathbf{L}_2$. Our goal is to compute the fixpoint of h by using the fixpoints of f and g. We start with a simple case:

Lemma 3.2. *Assume that g does not depend on the first argument, i.e., $g : \mathbf{L}_2 \to \mathbf{L}_2$. If p, q are two numbers such that g is q-stable and, denoting $\bar{y} \stackrel{\text{def}}{=} g^{(q)}(\bot_2)$, the function $F(x) \stackrel{\text{def}}{=} f(x, \bar{y})$ is p-stable, then $\mathsf{lfp}(h)$ exists and is equal to $\mathsf{lfp}(h) = (\bar{x}, \bar{y})$ where $\bar{x} \stackrel{\text{def}}{=} F^{(p)}(\bot_1)$. Moreover, h is $p + q$-stable.*

Proof. We verify that $(\bar{x}, \bar{y})$ is a fixpoint of h, by direct calculation: $h(\bar{x}, \bar{y}) = (f(\bar{x}, \bar{y}), g(\bar{y})) = (F(\bar{x}), g(\bar{y})) = (\bar{x}, \bar{y})$.

Next, we show that the stability index of h is at most $p + q$. For convenience, define:

$$\begin{aligned} y_0 &\stackrel{\text{def}}{=} \bot_2, & \forall \ell \geq 0 :\quad y_{\ell+1} &\stackrel{\text{def}}{=} g(y_\ell), \\ x_0 &\stackrel{\text{def}}{=} \bot_1, & \forall k \geq 0 :\quad x_{k+1} &\stackrel{\text{def}}{=} f(x_k, y_q), \\ (a_0, b_0) &\stackrel{\text{def}}{=} (\bot_1, \bot_2), & \forall n \geq 0 :\quad (a_{n+1}, b_{n+1}) &\stackrel{\text{def}}{=} h(a_n, b_n). \end{aligned}$$

By definition, $\bar{x} = x_p$ and $\bar{y} = y_q$. We claim the following three statements:

$$\forall n :\quad (a_n, b_n) \sqsubseteq (\bar{x}, \bar{y}), \tag{18}$$

$$\forall \ell \in \{0, \ldots, q\} :\quad (\bot_1, y_\ell) \sqsubseteq (a_\ell, b_\ell), \tag{19}$$

$$\forall k \in \{0, \ldots, p\} :\quad (x_k, y_q) \sqsubseteq (a_{q+k}, b_{q+k}). \tag{20}$$

Assuming the claims hold, by setting $k = p$ in Equation (20), we get $(\bar{x}, \bar{y}) = (x_p, y_q) \sqsubseteq (a_{p+q}, b_{p+q})$, which, together with inequality (18), proves that $(a_{p+q}, b_{p+q}) = (\bar{x}, \bar{y})$ and that h is $p + q$-stable.

Now, we prove the claims.

- Equation (18) is immediate, because $(\bar{x}, \bar{y})$ is a fixpoint of h, and $(a_n, b_n) = h^{(n)}(\bot_1, \bot_2)$ is below any fixpoint of h (due to monotonicity of h).
- To prove Equation (19), we claim that $y_\ell = b_\ell$ for all ℓ. Indeed, $(a_{\ell+1}, b_{\ell+1}) = h(a_\ell, b_\ell) = (f(a_\ell, b_\ell), g(b_\ell))$, which implies $b_{\ell+1} = g(b_\ell)$, which means that y_ℓ and b_ℓ are the same sequence.
- Finally, we show Equation (20) by induction on k. The base case, $k = 0$, follows from $(x_0, y_q) = (\bot_1, y_q) \sqsubseteq (a_q, b_q)$ by Equation (19). Assuming the claim holds for k, we have $(x_{k+1}, y_q) = (f(x_k, y_q), y_q) \sqsubseteq (f(a_{q+k}, b_{q+k}), b_{q+k}) \sqsubseteq (f(a_{q+k}, b_{q+k}), g(b_{q+k})) = h(a_{q+k}, b_{q+k}) = (a_{q+k+1}, b_{q+k+1})$, proving that the claim holds for $k + 1$. □

Next, we generalize the result to the case when both functions depend on both inputs: $f : \mathbf{L}_1 \times \mathbf{L}_2 \to \mathbf{L}_1$ and $g : \mathbf{L}_1 \times \mathbf{L}_2 \to \mathbf{L}_2$. We prove:

$$\begin{tikzcd}
(x_0 \overset{\text{def}}{=} \bot_1, y_{0,0} \overset{\text{def}}{=} \bot_2) \arrow[r, "g"] & (x_0, y_{0,1}) \arrow[r, "g"] & \cdots & (x_0, y_{0,q}) \arrow[d, "f"] & & & & & \\
& & & (x_1, y_{1,0} \overset{\text{def}}{=} \bot_2) \arrow[r, "g"] & (x_1, y_{1,1}) \arrow[r, "g"] & \cdots & & & \\
& & & & & \cdots & \cdots \arrow[d, "f"] & & \\
& & & & & & (x_p, y_{p,0} \overset{\text{def}}{=} \bot_2) \arrow[r, "g"] & \cdots \arrow[r, "g"] & (x_p, y_{p,q})
\end{tikzcd}$$

Fig. 1. Computing the fixpoint of (f, g).

Lemma 3.3. *Assume that p, q are two numbers such that, for each $u \in \mathbf{L}_1$, the function $g_u(y) \overset{def}{=} g(u, y)$ is q-stable, and the function $F(x) \overset{def}{=} f(x, g_x^{(q)}(\bot_2))$ is p-stable. Then, the following hold:*

(1) The function h is $(pq + p + q)$-stable and has the least fixpoint $(\bar{x}, \bar{y})$, where

$$\bar{x} \overset{def}{=} F^{(p)}(\bot_1) \qquad\qquad \bar{y} \overset{def}{=} g_{\bar{x}}^{(q)}(\bot_2). \tag{21}$$

(2) Further assume that f, g also satisfy the symmetric conditions: $f_v(x) \overset{def}{=} f(x, v)$ is p-stable for all $v \in \mathbf{L}_2$, and $G(y) \overset{def}{=} g(f_y^{(p)}(\bot_1), y)$ is q-stable. Then, denoting $(a_n, b_n) \overset{def}{=} h^{(n)}(\bot_1, \bot_2)$, the following equalities hold:

$$a_{pq+p} = \bar{x} \qquad\qquad b_{pq+q} = \bar{y}. \tag{22}$$

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

Proof. We start by proving Item (1), generalizing the proof of Lemma 3.2. To verify that $(\bar{x}, \bar{y})$ is a fixpoint of h, we need to show that $f(\bar{x}, \bar{y}) = \bar{x}$ and $g(\bar{x}, \bar{y}) = \bar{y}$. Indeed, these follow from q-stability of $g_{\bar{x}}$ and p-stability of F:

$$g(\bar{x}, \bar{y}) = g_{\bar{x}}(\bar{y}) = g_{\bar{x}}(g_{\bar{x}}^{(q)}(\bot_2)) = g_{\bar{x}}^{(q+1)}(\bot_2) = g_{\bar{x}}^{(q)}(\bot_2) = \bar{y},$$
$$f(\bar{x}, \bar{y}) = f(\bar{x}, g_{\bar{x}}^{(q)}(\bot_2)) = F(\bar{x}) = F(F^{(p)}(\bot_1)) = F^{(p)}(\bot_1) = \bar{x}.$$

To complete the proof of Item (1), we next prove that h is $(pq + p + q)$-stable. To this end, define the following sequences:

$$\begin{aligned}
x_0 &\overset{\text{def}}{=} \bot_1, & \forall k \in \{0, \ldots, p-1\} :\ \ x_{k+1} &\overset{\text{def}}{=} f(x_k, y_{k,q}), \\
y_{k,0} &\overset{\text{def}}{=} \bot_2, & \forall \ell \in \{0, \ldots, q-1\} :\ \ y_{k,\ell+1} &\overset{\text{def}}{=} g(x_k, y_{k,\ell}), \\
(a_0, b_0) &\overset{\text{def}}{=} (\bot_1, \bot_2), & \forall n \geq 0 :\ \ (a_{n+1}, b_{n+1}) &\overset{\text{def}}{=} h(a_n, b_n).
\end{aligned}$$

The sequences x_k and $y_{k,\ell}$ are illustrated in Figure 1. We claim that the sequences satisfy the following two properties:

$$\forall n :\ \ (a_n, b_n) \sqsubseteq (\bar{x}, \bar{y}), \tag{23}$$

$$\forall k \in \{0, \ldots, p\}, \forall \ell \in \{0, \ldots, q\} :\ \ (x_k, y_{k,\ell}) \sqsubseteq (a_n, b_n) \qquad \text{where } n = k(q+1) + \ell. \tag{24}$$

Before proving them, we show how they help complete the proof of both (1) and (2).

- By setting $(k,\ell) \stackrel{\text{def}}{=} (p,q)$ and $n = p(q+1)+q = pq+p+q$ in Equation (24), we obtain $(\bar{x},\bar{y}) = (x_p, y_{p,q}) \sqsubseteq (a_n, b_n)$, which, together with Equation (23) proves that $(\bar{x},\bar{y}) = (a_n,b_n)$ and, therefore, h is $pq+p+q$-stable. This completes the proof of Item (1) of the lemma.
- For Item (2), we notice that if we set $(k,\ell) \stackrel{\text{def}}{=} (p,0)$ and $n = p(q+1) = pq+p$ in Equation (24), then we obtain $(\bar{x}, y_{p,0}) = (x_p, y_{p,0}) \sqsubseteq (a_n,b_n)$, which implies $\bar{x} = a_n = a_{pq+p}$. By switching the roles of f, g, we also obtain $\bar{y} = b_{pq+q}$, which proves the lemma. In particular, h is $pq + \max(p,q)$-stable.

We now prove the two claims (23) and (24). The first claim (23) is immediate, because $(\bar{x},\bar{y})$ is a fixpoint of h and $(a_n,b_n) = h^{(n)}(\bot_1,\bot_2)$ is below any fixpoint.

For the second claim in Equation (24), refer to Figure 1 for some intuition; in particular, note that the mapping $(k,\ell) \mapsto n \stackrel{\text{def}}{=} k(q+1)+\ell$ is injective for $k \in \{0,\dots,p\}$, $\ell \in \{0,\dots,q\}$, and n represents the position of $(x_k, y_{k,\ell})$ in the sequence defined in the figure. Note also that, $y_{k,q} = \bar{y}$ for all $k \in \{0,\dots,p\}$ and $x_p = \bar{x}$; and $a_n \sqsubseteq f(a_n,b_n)$ and $b_n \sqsubseteq g(a_n,b_n)$: This follows from h being monotone: $h^{(n)}(\bot_1,\bot_2) \sqsubseteq h^{(n+1)}(\bot_1,\bot_2)$. We prove Equation (24) by induction on n.

The base case when $n=0$ then $k=\ell=0$ is trivial, as both sides of Equation (24) are $(\bot_1,\bot_2)$. For the inductive case, assume $n>0$, and let k,ℓ be the unique values s.t. $n = k(q+1)+\ell$. Consider two cases, corresponding to whether we are taking a horizontal step or a vertical step in Figure 1:

Case 1: $\ell > 0$. Then, the pre-image of $n-1$ is $k, \ell-1$, in other words $(n-1) = k(q+1)+(\ell-1)$, and we have $(x_k, y_{k,\ell-1}) \sqsubseteq (a_{n-1}, b_{n-1})$ by induction hypothesis for $n-1$. It follows that

$$(x_k, y_{k,\ell}) = (x_k, g(x_k, y_{k,\ell-1})) \sqsubseteq (a_{n-1}, g(a_{n-1},b_{n-1})) \sqsubseteq (f(a_{n-1},b_{n-1}), g(a_{n-1},b_{n-1})) = (a_n,b_n),$$

where we used the induction hypothesis and the fact that $a_{n-1} \sqsubseteq f(a_{n-1},b_{n-1})$.

Case 2: $\ell = 0$. Then, the pre-image of $n-1$ is $(k-1,q)$, in other words $(n-1) = (k-1)(q+1)+q$, and we have $(x_{k-1}, y_{k-1,q}) \sqsubseteq (a_{n-1},b_{n-1})$ by induction hypothesis. It follows:

$$\begin{aligned}(x_k, y_{k,0}) = (x_k, \bot_2) = (f(x_{k-1}, y_{k-1,q}), \bot_2) &\sqsubseteq (f(x_{k-1}, y_{k-1,q}), y_{k-1,q}) \\ &\sqsubseteq (f(a_{n-1},b_{n-1}), b_{n-1}) \sqsubseteq (f(a_{n-1},b_{n-1}), g(a_{n-1},b_{n-1})) = (a_n,b_n).\end{aligned}$$

This completes the proof of Equation (24). □

Next, we present Theorem 3.4, which generalizes Lemma 3.3 from 2 functions to n functions. Recall that, in the lemma, we had to consider two derived functions g_u and F from f and g. When generalizing to n functions, the number of derived functions becomes unwieldy, and it is more convenient to state the theorem for a clone of functions. A *clone* [14] over n posets $\mathbf{L}_1,\dots,\mathbf{L}_n$ is a set of functions C where (1) each element $f \in C$ of the form $f : \mathbf{L}_{j_1} \times \cdots \times \mathbf{L}_{j_k} \to \mathbf{L}_i$, where $1 \le j_1 < j_2 < \cdots < j_k \le n$ and $1 \le i \le n$, is monotone, (2) C contains all projections $\mathbf{L}_{j_1} \times \cdots \times \mathbf{L}_{j_k} \to \mathbf{L}_{j_i}$, 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 all constant functions: $g_u : () \to \mathbf{L}_i$, $g_u() \stackrel{\text{def}}{=} u$, for every fixed $u \in L_i$.

For a simple illustration of a c-clone, consider a POPS $\boldsymbol{P}$, and define C to consist of all multivariate polynomials over variables $x_1,\dots,x_n$. More precisely, set $\mathbf{L}_1 = \cdots = \mathbf{L}_n \stackrel{\text{def}}{=} \boldsymbol{P}$, and, for all choices of indices $1 \le j_1 < j_2 < \cdots < j_k \le n$ and $1 \le i \le n$, let C contain all polynomials $f(x_{j_1},\dots,x_{j_k})$, viewed as functions $\mathbf{L}_{j_1} \times \cdots \times \mathbf{L}_{j_k} \to \mathbf{L}_i$. Then, C is a c-clone.

Theorem 3.4. *Let C be a c-clone of functions over n posets $\mathbf{L}_1,\dots,\mathbf{L}_n$, and assume that, for every $i \in [n]$ where we denote by $[n]$ the integers from 1 to n, every function $f : \mathbf{L}_i \to \mathbf{L}_i$ in C is p_i-stable. Assume w.l.o.g. that $p_1 \ge p_2 \ge \cdots \ge p_n$. Let $f_1,\dots,f_n$ be functions in C, where $f_i : \mathbf{L}_1 \times \cdots \times \mathbf{L}_n \to \mathbf{L}_i$,* *and define the function $h \overset{def}{=} (f_1, \dots, f_n)$. Then, h is p-stable, where $p \overset{def}{=} \sum_{k=1,n} \prod_{i=1,k} p_i$. Moreover, this upper bound is tight: There exist posets $\mathbf{L}_1, \dots, \mathbf{L}_n$, a c-clone C, and functions $f_1, \dots, f_n$, such that p is the stability index of h.*

Proof. We defer the lower bound to Appendix A and prove here the upper bound. It will be convenient to define the following expression, for every $m \geq 0$ and numbers $a_1, \dots, a_m$:

$$E_m(a_1, \dots, a_m) \overset{\text{def}}{=} a_1 + a_1 a_2 + a_1 a_2 a_3 \cdots + a_1 a_2 \cdots a_m = \sum_{i=1,m} \prod_{j=1,i} a_j.$$

Note that if we permute the sequence $a_1, \dots, a_m$, then the expression $E_m(a_1, \dots, a_m)$ is maximized when the sequence is decreasing, $a_1 \geq a_2 \geq \cdots \geq a_m$.

We prove by induction on n that h is $E_n(p_1, \dots, p_n)$-stable. When $n = 1$, then the statement holds vacuously, because a function $f : \mathbf{L}_1 \longrightarrow \mathbf{L}_1$ is p_1-stable by assumption. Assume $n > 1$ and let $f_1, \dots, f_n$ be as in the statement of the theorem.

Fix an arbitrary dimension $i \in [n]$. Let $\mathbf{L}_{-i} \overset{\text{def}}{=} \mathbf{L}_1 \times \cdots \times \mathbf{L}_{i-1} \times \mathbf{L}_{i+1} \cdots \times \mathbf{L}_n$ be the product of all posets other than $\mathbf{L}_i$. Given $\mathbf{x} = (x_1, \dots, x_n) \in \mathbf{L}_1 \times \cdots \times \mathbf{L}_n$, denote by

$$x_{-i} \overset{\text{def}}{=} (x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n) \in \mathbf{L}_{-i} \tag{25}$$

the vector consisting of all coordinates other than i. Define the functions $f : \mathbf{L}_i \times \mathbf{L}_{-i} \longrightarrow \mathbf{L}_i$ and $g : \mathbf{L}_i \times \mathbf{L}_{-i} \longrightarrow \mathbf{L}_{-i}$:

$$f(x_i, x_{-i}) \overset{\text{def}}{=} f_i(x) \qquad\qquad g(x_i, x_{-i}) \overset{\text{def}}{=} (f_1(x), \dots, f_{i-1}(x), f_{i+1}(x), \dots, f_n(x)).$$

Then, h can be written as $h(\mathbf{x}) = (f(x_i, x_{-i}), g(x_i, x_{-i}))$ (with some abuse, assuming $f(x_i, x_{-i})$ is moved from the 1st to the ith position). The assumptions of Lemma 3.3, including those of Item (2), are satisfied. For example, given $x_i \in \mathbf{L}_i$, the function $g_{x_i}(x_{-i}) \overset{\text{def}}{=} g(x_i, x_{-i})$ is in the c-clone C and has type $\mathbf{L}_{-i} \longrightarrow \mathbf{L}_{-i}$; hence, it is $q \overset{\text{def}}{=} E_{n-1}(p_1, \dots, p_{i-1}, p_{i+1}, \dots, p_n)$-stable, by induction hypothesis. Similarly, the function $F(x_i) \overset{\text{def}}{=} f(x_i, g_{x_i}^{(q)}(\bot))$ is in the c-clone C and has type $\mathbf{L}_i \longrightarrow \mathbf{L}_i$, hence, it is p_i-stable by the assumption of the theorem. The conditions for Item (2) of Lemma 3.3 are verified similarly.

Lemma 3.3 implies two things: First, h has a least fixpoint, denoted by $\mathsf{lfp}(h) = (\bar{x}_1, \dots, \bar{x}_n)$. Second, from Item (2) of the lemma and from induction hypothesis, the i-component of $h^{(r)}(\bot)$ reaches the fixpoint $(h^{(r)}(\bot))_i = \bar{x}_i$ when

$$r = p_i q + p_i = p_i \cdot E_{n-1}(p_1, \dots, p_{i-1}, p_{i+1}, \dots, p_n) + p_i \leq E_n(p_1, \dots, p_n).$$

Note that once the fixpoint in a dimension is reached, it stays fixed. Since i was arbitrary, h reaches the fixpoint in all dimensions after $E_n(p_1, \dots, p_n)$ iterations. □

4 DATALOG°

We define here the language datalog°, which generalizes datalog from traditional relations to $\boldsymbol{P}$-relations, for some POPS $\boldsymbol{P}$. As in datalog, the input relations to the program will be called **Extensional Database Predicates (EDB)**, and the computed relations will be called **Intensional Database Predicates (IDB)**. Each EDB can be either a $\boldsymbol{P}$-relation or standard relation, i.e., a $\mathbb{B}$-relation, and we denote by $\sigma \overset{\text{def}}{=} \{R_1, \dots, R_m\}$ and $\sigma_{\mathbb{B}} \overset{\text{def}}{=} \{B_1, \dots, B_k\}$ the two vocabularies. All IDBs are $\boldsymbol{P}$-relations, and their vocabulary is denoted by $\tau = \{T_1, \dots, T_n\}$.

A datalog$^\circ$ program Π consists of n sum-sum-product rules $r_1, \ldots, r_n$ (as in Definition 2.7), where each rule r_i has the IDB T_i in the head:

$$\begin{aligned} r_1 : \quad & T_1(\cdots) \text{ :- } E_{11} \oplus E_{12} \oplus \cdots \\ & \qquad \cdots \\ r_n : \quad & T_n(\cdots) \text{ :- } E_{n1} \oplus E_{n2} \oplus \cdots , \end{aligned} \tag{26}$$

and each E_{ij} is a sum-product expression as in Equation (10). The program Π is said to be *linear* if each sum-product expression E_{ij} contains at most one IDB predicate.

4.1 Least Fixpoint of the Immediate Consequence Operator

The ***Immediate Consequence Operator*(ICO)** associated to a program Π is the function $F : \mathsf{Inst}(\sigma, D, \boldsymbol{P}) \times \mathsf{Inst}(\sigma_{\mathbb{B}}, D, \mathbb{B}) \times \mathsf{Inst}(\tau, D, \boldsymbol{P}) \longrightarrow \mathsf{Inst}(\tau, D, \boldsymbol{P})$, which takes as input an instance $(I, I_{\mathbb{B}})$ of the EDBs and an instance J of the IDBs and computes a new instance $F(I, I_{\mathbb{B}}, J)$ of the IDBs by evaluating each sum-sum-product rule. By fixing the EDBs, we will view the ICO as function from IDBs to IDBs, written as $F(J)$. We define the *semantics* of the datalog$^\circ$ program (26) as the least fixpoint of the ICO F, when it exists.

ALGORITHM 1: Naïve Evaluation for datalog$^\circ$

```
$J^{(0)} \leftarrow \bot$;                // In a naturally ordered semiring this becomes $J^{(0)} \leftarrow 0$
for $t \leftarrow 0$ to $\infty$ do
    $J^{(t+1)} \leftarrow F(J^{(t)})$;
    if $J^{(t+1)} = J^{(t)}$ then
        Break
return $J^{(t)}$
```

The naïve algorithm for evaluating datalog$^\circ$ is shown in Algorithm 1, and it is quite similar to that for standard, positive datalog with set semantics. We start with all IDBs at $\bot$, then repeatedly apply the ICO F until we reach a fixpoint. The algorithm computes the increasing sequence $\bot \sqsubseteq F(\bot) \sqsubseteq F^{(2)}(\bot) \sqsubseteq \cdots$ When the algorithm terminates, we say that it *converges*; in that case it returns the least fixpoint of F. Otherwise, we say that the algorithm *diverges*.

4.2 Convergence of datalog$^\circ$ Programs

While every pure datalog program is guaranteed to have a least fixpoint, this no longer holds for datalog$^\circ$ programs. As we mentioned in the introduction, there are five possibilities, depending on the POPS $\boldsymbol{P}$:

(i) $\bigvee_t J^{(t)}$ is not a fixpoint of the ICO; in this case, we say that the program *diverges*. For example, suppose the POPS consists of $\mathbb{N} \times \mathbb{N}$, with pairwise addition and multiplication, i.e., $(x, y) \oplus (u, v) = (x + u, y + v)$ and similarly for $\otimes$, and with the lexicographic order $(x, y) \sqsubseteq (u, v)$ defined as $x < u$ or $x = u$ and $y \leq v$. If the ICO of a program is the function $F(x, y) = (x, y + 1)$, then $\bigvee_{t \geq 0} F^{(t)}(0, 0) = \bigvee_{t \geq 0}(0, t) = (1, 0)$ is not a fixpoint of F, because $F(1, 0) = (1, 1)$; in fact, F has no fixpoint.

(ii) $\bigvee_n J^{(n)}$ is always the least fixpoint, but the naïve algorithm does not always terminate. With some abuse, we say also in this case that the program *diverges*. For a simple example, consider the semiring $\mathbb{N} \cup \{\infty\}$ and the function $F(x) = x + 1$. Its fixpoint is ∞, but it is not computable in a finite number of steps.

(iii) The naïve algorithm always terminates, in which case, we say that it *converges*. The number of steps depends on the input EDB database, meaning *both* the number of ground atoms in the EDB, and their values in $\boldsymbol{P}$.

(iv) The naïve algorithm always terminates in a number of steps that depends only on the number of ground atoms in the EDB, but not on their values.

(v) The naïve algorithm always terminates in a number of steps that is polynomial in the number of ground atoms in the EDB.

In this article, we are interested in characterizing the POPS that ensure that every datalog$^\circ$ program converges. At a finer level, our goal is to characterize precisely cases iii–v. We will do this in Section 5, and, for that purpose, we will use an equivalent definition of the semantics of a datalog$^\circ$ program, namely, as the least fixpoint of a tuple of polynomials, obtained by grounding the program.

4.3 Least Fixpoint of the Grounded Program

In this article, we consider an equivalent semantics of datalog$^\circ$, which consists of first grounding the program, then computing its least fixpoint.

Fix an EDB instance $I, I_{\mathbb{B}}$, and let $D_0 \subseteq D$ be the finite set consisting of its active domain plus all constants occurring in the program Π. Let $M = |\mathsf{GA}(\sigma, D_0)|$ and $N = |\mathsf{GA}(\tau, D_0)|$ be the number of ground tuples of the EDBs and the IDBs, respectively. We associate them in 1-to-1 correspondence with $M + N$ POPS variables $z_1, \ldots, z_M$ and $x_1, \ldots, x_N$, and use the same notation as in Section 2.4 by writing $x_{T_i(\mathbf{a})}$ for the variable associated to the ground tuple $T_i(\mathbf{a})$.

Consider a rule $T_i(\cdots) \text{ :- } E_{i1} \oplus E_{i2} \oplus \cdots$ of the datalog$^\circ$ program, with head relation T_i. A *grounding* of this rule is a rule of the form:

$$x_{T_i(\mathbf{a})} \text{ :- } f_{T_i(\mathbf{a})}(z_1, \ldots, z_M, x_1, \ldots, x_N),$$

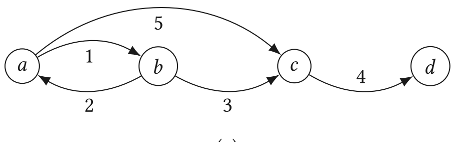
where $T_i(\mathbf{a}) \in \mathsf{GA}(T_i, D_0)$ is a ground tuple, and $f_{T_i(\mathbf{a})}$ is the *provenance polynomial* (defined in Section 2.4) of the rule's body $E_{i1} \oplus E_{i2} \oplus \cdots$. Since the value of each EDB variable $z_{R_i(\mathbf{u})}$ is known, we can substitute it with its value, and the provenance polynomial simplifies to one that uses only IDB variables x_j; we will no longer refer to the EDB variables z_i. The *grounded program* consists of all N groundings, of all rules. Using more friendly indexes, we write the grounded program as:

$$\begin{aligned} x_1 &\text{ :- } f_1(x_1, \ldots, x_N) \\ &\cdots \\ x_N &\text{ :- } f_N(x_1, \ldots, x_N), \end{aligned} \tag{27}$$

where each f_i is a multivariate polynomial in the variables $x_1, \ldots, x_N$. We write $\mathbf{f} = (f_1, \ldots, f_N)$ for the vector-valued function whose components are the N provenance polynomials and define the semantics of the datalog$^\circ$ program as its least fixpoint, $\mathsf{lfp}(\mathbf{f})$, when it exists, where, as usual, we identify the tuple $\mathsf{lfp}(\mathbf{f}) \in \boldsymbol{P}^N$ with an IDB instance $\mathsf{lfp}(\mathbf{f}) \in \mathsf{Inst}(\tau, D_0, \boldsymbol{P})$. By definition, $\mathsf{lfp}(\mathbf{f})$ is equal to the least fixpoint of the ICO, as defined in Section 4.1.

4.4 Examples

We illustrate datalog$^\circ$ with two examples. When the POPS $\boldsymbol{P}$ is a naturally ordered semiring, then we will use the following *indicator function* $[C]_0^1$, which maps a Boolean condition C to either $0 \in \boldsymbol{P}$ or $1 \in \boldsymbol{P}$, depending on whether C is false or true. We write the indicator function simply as $[C]$, when the values 0,1 are clear from the context. An indicator function can be desugared by replacing $\{[C] \otimes P_1 \otimes \cdots \otimes P_k \mid \Phi\}$ with $\{P_1 \otimes \cdots \otimes P_k \mid \Phi \wedge C\}$. When $\boldsymbol{P}$ is not naturally ordered, then we will not use indicator functions; see Example 2.6.



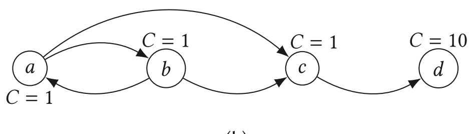


Fig. 2. A graph illustrating Example 4.1 (a) and Example 4.2 (b).

Example 4.1. Let the EDB and IDB vocabularies be $\sigma = \{E\}$ and $\tau = \{L\}$, where E is binary and L is unary. Consider the following datalog$^\circ$ program:

$$L(X) \text{ :- } [X = a] \oplus \bigoplus_{Z} (L(Z) \otimes E(Z,X)), \tag{28}$$

where $a \in D$ is some constant in the domain. We show three different interpretations of the program over three different naturally ordered semirings. First, we interpret it over the semiring of Booleans. In this case, the program can be written in a more familiar notation:

$$L(X) \text{ :- } [X = a]_0^1 \vee \exists_Z (L(Z) \wedge E(Z,X)).$$

This is the reachability program, which computes the set of nodes X reachable from the node a. The indicator function $[X = a]_0^1$ returns 0 or 1, depending on whether $X \neq a$ or $X = a$.

Next, let us interpret it over Trop$^+$. In that case, the indicator function $[X = a]_\infty^0$ returns ∞ when $X \neq a$, and returns 0 when $X = a$. The program becomes:

$$L(X) \text{ :- } \min\left((\texttt{if } X = a \texttt{ then 0 else } \infty), \min_Z(L(Z) + E(Z,X))\right).$$

This program solves the **Single-Source-Shortest-Path (SSSP)** problem with source vertex a. Consider the graph in Figure 2(a). The active domain consists of the constants a, b, c, d, and the naïve evaluation algorithm converges after 5 steps, as shown here:

	$L(a)$	$L(b)$	$L(c)$	$L(d)$
$L^{(0)}$	∞	∞	∞	∞
$L^{(1)}$	0	∞	∞	∞
$L^{(2)}$	0	1	5	∞
$L^{(3)}$	0	1	4	9
$L^{(4)}$	0	1	4	8
$L^{(5)}$	0	1	4	8

Third, let us interpret it over Trop$_p^+$, defined in Example 2.9. Assume for simplicity that $p = 1$. In that case the program computes, for each node X, the bag $\{\{l_1, l_2\}\}$ of the lengths of the two shortest paths from a to X. The indicator function $[X = a]$ is equal to $\{\{0, \infty\}\}$ when $X = a$, and equal to $\{\{\infty, \infty\}\}$ otherwise. The reader may check that the program converges to:

$$L(a) = \{\{0, 3\}\} \qquad L(b) = \{\{1, 4\}\} \qquad L(c) = \{\{4, 5\}\} \qquad L(d) = \{\{8, 9\}\}.$$

Finally, we can interpret it over Trop$_{\leq\eta}^+$, the semiring in Example 2.10. In that case the program computes, for each X, the set of all possible lengths of paths from a to X that are no longer than the shortest path plus η.

Example 4.2. A classic problem that requires the interleaving of recursion and aggregation is the bill-of-material (see, e.g., Reference [87]), where we are asked to compute, for each part X, the total cost of X, of all sub-parts of X, all sub-sub-parts of X, and so on. The EDB and IDB schemas are $\sigma_{\mathbb{B}} = \{E\}$, $\sigma = \{C\}$, $\tau = \{T\}$. The relation $E(X,Y)$ is a standard, Boolean relation, representing the fact that "X has a subpart Y"; $C(X)$ is an $\mathbb{N}$-relation or an $\mathbb{R}_\perp$-relation (to be discussed shortly) representing the cost of X; and $T(X)$ is the *total cost* of X, which includes the cost of all its (sub-)subparts. The datalog$^\circ$ program is:

$$T(X) \text{ :- } C(X) + \sum_Y \{T(Y) \mid E(X,Y)\}.$$

When the graph defined by E is a tree, then the program computes correctly the bill-of-material. We are interested, however, in what happens when the graph encoded by E has cycles, as illustrated in Figure 2(b). The grounded program is[5]:

$$\begin{aligned} T(a) &\text{ :- } C(a) + T(b) + T(c) \\ T(b) &\text{ :- } C(b) + T(a) + T(c) \\ T(c) &\text{ :- } C(c) + T(d) \\ T(d) &\text{ :- } C(d). \end{aligned}$$

We consider two choices for the POPS. First, the naturally ordered semiring $(\mathbb{N}, +, *, 0, 1)$. Here, the program diverges, since the naïve algorithm will compute ever increasing values for $T(a)$ and $T(b)$, which are on a cycle. Second, consider the lifted reals $\mathbb{R}_\perp = (\mathbb{R} \cup \{\perp\}, +, *, 0, 1, \sqsubseteq)$. Now, the program converges in three steps, as can be seen below:

	$T(a)$	$T(b)$	$T(c)$	$T(d)$
T_0	$\perp$	$\perp$	$\perp$	$\perp$
T_1	$\perp$	$\perp$	$\perp$	10
T_2	$\perp$	$\perp$	11	10
T_3	$\perp$	$\perp$	11	10

4.5 Extensions

We discuss here several extensions of datalog$^\circ$ that we believe are needed in a practical implementation.

Case Statements Sum-products can be extended w.l.o.g. to include case statements of the form:

$$T(x_1,\ldots,x_k) \text{ :- } \texttt{case } C_1 : E_1; \quad C_2 : E_2; \cdots ;[\texttt{ else } E_n],$$

where $C_1, C_2, \ldots$ are conditions and $E_1, E_2, \ldots$ are sum-product expressions. This can be desugared to a sum-sum-product:

$$T(x_1,\ldots,x_k) \text{ :- } \{E_1 \mid C_1\} \oplus \{E_2 \mid \neg C_1 \wedge C_2\} \oplus \cdots \oplus \{E_n \mid \neg C_1 \wedge \neg C_2 \cdots\}$$

and therefore the least fixpoint semantics and our convergence results in Section 5 continue to hold. For example, we may compute the prefix-sum of a vector V of length 100 as follows:

$$W(i) \text{ :- } \texttt{case } i = 0 : V(0); \quad i < 100 : W(i-1) + V(i).$$

Multiple Value Spaces. Our discussion so far assumed that all rules in a datalog$^\circ$ program are over a single POPS. In practice, one often wants to perform computations over multiple POPS. In that case, we need to have some predefined functions mapping between various POPS; if these are monotone, then the least fixpoint semantics still applies, otherwise, the program needs to be *stratified*. We illustrate with an example, which uses two POPS: $\mathbb{R}_+$ and $\mathbb{B}$.

[5]Strictly speaking, we should have introduced POPS variables, $x_{T(a)}, x_{T(b)}, \ldots$, but, to reduce clutter, we show here directly the grounded atoms instead of their corresponding POPS variable.

Example 4.3. We illustrate the *company control* example from Reference [71, Example 3.2]. $S(X,Y) = n \in \mathbb{R}_+$ represents the fact that company X owns a proportion of n shares in company Y. We want to compute the predicate $C(X,Y)$, representing the fact that the company X *controls* company Y, where control is defined as follows: X controls Y if the sum of shares it owns in Y plus the sum of shares in Y owned by companies controlled by X is > 0.5. The program is adapted directly from Reference [71]:

$$\begin{aligned} CV(X,Z,Y) &\coloneq [X = Z] * S(X,Y) + [C(X,Z)] * S(Z,Y) \\ T(X,Y) &\coloneq \sum_Z \{CV(X,Z,Y) \mid \text{Company}(Z)\} \\ C(X,Y) &\coloneq [T(X,Y) > 0.5]. \end{aligned}$$

The value of $CV(X,Z,Y)$ is the fraction of shares that X owns in Y through its control of company Z; when $X = Z$, then this fraction includes $S(X,Y)$. The value of $T(X,Y)$ is the total amount of shares that X owns in Y. The last rule checks whether this total is > 0.5: In that case, X controls Y.

The EDB and IDB vocabularies are $\sigma = \{S\}$, $\sigma_{\mathbb{B}} = \{\text{Company}\}$, $\tau = \{CV, T\}$, $\tau_{\mathbb{B}} = \{C\}$. The IDBs CV, T are $\mathbb{R}_+$-relations, C is a standard $\mathbb{B}$-relation. The mapping between the two POPS is achieved by the indicator function $[\Phi] \in \mathbb{R}_+$, which returns 0 when the predicate Φ is false and 1 otherwise. All rules are monotone, w.r.t. to the natural orders on $\mathbb{R}_+$ and $\mathbb{B}$, and, thus, the least-fixpoint semantics continues to apply to this program. But the results in Section 5 apply only to fixpoints of polynomials, while the grounding of our program is no longer a polynomial due to the use of the indicator functions.

Interpreted functions over the key-space. A practical language needs to allow interpreted functions over the key space, i.e., the domain D, as illustrated by this simple example:

$$\texttt{Shipping}(cid, \texttt{date} + 1) \coloneq \texttt{Order}(cid, \texttt{date}).$$

Here, $\texttt{date} + 1$ is an interpreted function applied to $\texttt{date}$. Interpreted functions over D may cause the active domain to grow indefinitely, leading to divergence; our results in Section 5 apply only when the active domain is fixed, enabling us to define the grounding (27) of the datalog$^\circ$ program.

Keys to Values. Finally, a useful extension is to allow key values to be used as POPS values when the types are right. For example, if $\texttt{Length}(X,Y,C)$ is Boolean relation, where a tuple (X,Y,C) represents the fact that there exists a path of length C from X to Y, then we can compute the length of the shortest path as the following rule of the tropical semiring Trop^+:

$$\texttt{ShortestLength}(X,Y) \coloneq \min_C \left([\texttt{Length}(X,Y,C)]^0_\infty + C\right).$$

The key variable C became an atom over the tropical semiring.

5 CHARACTERIZING THE CONVERGENCE OF datalog$^\circ$

In this section, we prove our main result, Theorem 1.2. As we saw in Section 1 (and again in Section 4.2), there are five different possibilities for the divergence/convergence of datalog$^\circ$ programs. Our results in this section concern the last three cases, when every datalog$^\circ$ program converges. Recall that, by "converge," we mean that the naïve algorithm terminates in a finite number of steps; we are not interested in "convergence in the limit."

Throughout this section, we will assume that the datalog$^\circ$ program has been grounded, as in Section 4.3, and therefore our task is to study the convergence of a tuple of polynomials; see (27).

To reduce clutter when writing polynomials, we will follow the convention in Section 2.2 and use the symbols $+, \cdot$ instead of $\oplus, \otimes$, while keeping in mind that they represent abstract operations in a POPS.

Consider the following grounded datalog$^\circ$ program, with a single variable x and a single rule:

$$x \text{ :- } 1 + cx. \tag{29}$$

We need to compute the least fixpoint of $f(x) := 1 + cx$. When the POPS $\boldsymbol{P}$ is a naturally ordered semiring, then its smallest element is $\bot = 0$ and the naïve algorithm computes the sequence:

$$f^{(0)}(0) := 0 \quad f^{(1)}(0) := 1 \quad f^{(2)}(0) := 1 + c \quad f^{(3)}(0) := 1 + c + c^2 \quad \dots \quad f^{(q)}(0) := 1 + c + \cdots + c^{q-1}.$$

If we want *every* datalog$^\circ$ program to converge on $\boldsymbol{P}$, then surely so must our program (29). In essence, the main result in this section is that the converse holds, too: If (29) converges on $\boldsymbol{P}$ for any choice of c, then every datalog$^\circ$ program converges on $\boldsymbol{P}$. For example, (29) diverges on the semiring $\mathbb{N}$, because, if $c = 2$, then $f^{(q)}(0) = 1 + 2 + 2^2 + \cdots + 2^{q-1} \to \infty$. However, this program converges on the semiring Trop$^+$ and, therefore, *every* datalog$^\circ$ program converges on Trop$^+$.

In the rest of this section, we prove Theorem 1.2. We start by assuming that the POPS $\boldsymbol{P}$ is a naturally ordered semiring $\boldsymbol{S}$, then extend the result to arbitrary POPS.

5.1 Definition of Stable Semirings

The following notations and simple facts can be found in Reference [33]. Fix a semiring $\boldsymbol{S}$. For every $c \in \boldsymbol{S}$ and $p \geq 0$ define:

$$c^{(p)} \overset{\text{def}}{=} 1 + c + c^2 + \cdots + c^p. \tag{30}$$

Definition 5.1. An element $c \in \boldsymbol{S}$ is *p-stable* if $c^{(p)} = c^{(p+1)}$. We say that c is *stable* if there exists p such that c is p-stable.

We call the semiring $\boldsymbol{S}$ *stable* if every element is stable. We call it *uniformly stable* if there exists $p \geq 0$ such that every $c \in \boldsymbol{S}$ is p-stable; in that case, we also call $\boldsymbol{S}$ a *p-stable semiring*.

Note that an equivalent definition of c being p-stable is that

$$c^{(p)} = c^{(q)} \qquad \text{for all } q > p, \tag{31}$$

this can be seen by induction and from the fact that $c^{(q)} = 1 + c \cdot c^{(q-1)}$.

We recall here a very nice and useful result from Reference [33]. A short proof is supplied for completeness.

Proposition 5.2. *If 1 is p-stable, then $\boldsymbol{S}$ is naturally ordered. In particular, every stable semiring is naturally ordered.*

Proof. If 1 is p-stable, then $1^{(p+1)} = 1^{(p)}$, which means that $1 + 1 + \cdots + 1(p + 1 \text{ times}) = 1 + \cdots + 1(p \text{ times})$, or, in short, $p + 1 = p$. We prove that, if $a \leq_S b$ and $b \leq_S a$ hold, then $a = b$. By definition of $\leq_S$, there exist x, y such that $a + x = b$ and $b + y = a$. On one hand, we have $a = b + y = a + x + y$, which implies $a = a + k(x + y)$ for every $k \geq 0$, in particular, $a = a + p(x + y)$. On the other hand, we have $b = a + x = a + k(x + y) + x = a + (k + 1)x + ky$. We set $k = p$ and obtain $b = a + (p + 1)x + py = a + px + py = a + p(x + y)$ and the latter we have seen is $= a$, proving that $a = b$. □

The converse does not hold in general, for example, the semiring $\mathbb{N}$ is naturally ordered but is not stable. However, if $\boldsymbol{S}$ is both naturally ordered and satisfies the ACC condition (see Section 3), then it is also stable. Here, too, the converse fails: Trop$^+$ is 0-stable, because $\min(0, x) = 0$, yet it does not satisfy the ACC condition, e.g., the following is an infinitely ascending chain in Trop^+: $1 > 1/2 > 1/3 > 1/4 > \cdots$

The case of a 0-stable semiring has been most extensively studied. Such semirings are called *simple* by Lehmann [52], are called *c-semirings* by Kohlas [47], and *absorptive* by Dannert et al. [15]. In all cases, the authors require $1 + a = 1$ for all a (or, equivalently, $b + ab = b$ for all a, b [15]), which is equivalent to stating that a is 0-stable and also equivalent to stating that $(S, +)$ 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 $\cdot = \wedge$.

We give next two examples of stable semirings. The first one is p-stable for some $p > 0$ and it is not $(p-1)$-stable. The second one is non-uniformly stable. These examples are adapted from Reference [33].

Proposition 5.3. *The semiring* Trop^+_p *defined in Example 2.9 is p-stable. Moreover, the bound p on the stability of* Trop^+_p *is tight.*

Proof. Let $c \in \mathrm{Trop}^+_p$; recall that c is a bag of $p+1$ elements in $\mathbb{R}_+ \cup \{\infty\}$. For every $q \geq 1$, c^q consists of the $p+1$ smallest sums of the form $u_{i_1} + u_{i_2} + \cdots + u_{i_q}$, where each $u_{i_j} \in c$. Furthermore, the value $c^{(q)} = 1 + c + c^2 + \cdots + c^q$ is obtained as follows: (a) consider all sums of up to q numbers in c, and (b) retain the smallest $p+1$ sums, including duplicates. We claim that $c^{(p)} = c^{(p+1)}$. To prove the claim, consider such a sum $y \overset{\text{def}}{=} u_{i_0} + u_{i_1} + \cdots + u_{i_p}$ in c^{p+1}. This term is greater than or equal to each of the following $p+1$ terms: $0, u_{i_0}, u_{i_0} + u_{i_1}, \ldots, u_{i_0} + \cdots + u_{i_{p-1}}$, which are already included in the bags $1, c, c^2, \ldots, c^p$. This proves that every new term in c^{p+1} is redundant, proving that $c^{(p)} = c^{(p+1)}$.

To show that the bound p on the stability of Trop^+_p is tight, it suffices to show that the 1-element $\mathbf{1}_p = \{\!\{0, \infty, \ldots, \infty\}\!\}$ (i.e., p-times ∞) is not $(p-1)$-stable. Indeed, $\mathbf{1}_p^i = \mathbf{1}_p$ for every $i \geq 1$ and $\mathbf{1}_p + \mathbf{1}_p^1 + \mathbf{1}_p^2 + \cdots + \mathbf{1}_p^{p-1} = \{\!\{0, \ldots, 0, \infty\}\!\}$ (i.e., p-times 0) whereas $\mathbf{1}_p + \mathbf{1}_p^1 + \mathbf{1}_p^2 + \cdots + \mathbf{1}_p^p = \{\!\{0, \ldots, 0, 0\}\!\}$ (i.e., $p+1$-times 0). □

Proposition 5.4. *The semiring* $\mathrm{Trop}^+_{\leq\eta}$ *defined in Example 2.10 is stable. Moreover,* $\mathrm{Trop}^+_{\leq\eta}$ *is not p-stable for any p.*

Proof. The proof is similar to that of Proposition 5.3. While the elements of Trop^+_p are bags, those of $\mathrm{Trop}^+_{\leq\eta}$ are sets. Let c be an element of $\mathrm{Trop}^+_{\leq\eta}$. Then, $c^{(q)} = 1_{\leq\eta} + c + c^2 + \cdots + c^q$ is obtained as follows: (a) consider all sums of $\leq q$ numbers; this includes the empty sum, whose value is 0. (b) retain only those sums that are $\leq \eta$. If $c = \{0\}$, then c is 0-stable, so assume w.l.o.g. that c contains some element > 0. Let $x_0 > 0$ be the smallest such element, and let $p = \lceil \frac{\eta}{x_0} \rceil$. We claim that $c^{(p)} = c^{(p+1)}$. To prove the claim, consider a sum of $p+1$ elements $y = x_{i_0} + \cdots + x_{i_p}$ that belongs to $c^{(p+1)}$ but does not belong to $c^{(p)}$. In particular, $x_{i_j} \neq 0$, otherwise, we could drop x_{i_j} from the sum and we had $y \in c^{(p)}$. It follows that $y \geq (p+1)x_0 > \eta$, which means that y is not included in $c^{(p+1)}$. It follows that $c^{(p)} = c^{(p+1)}$, as required.

To show that $\mathrm{Trop}^+_{\leq\eta}$ is not p-stable for any p, we assume to the contrary that there does exist p such that $\mathrm{Trop}^+_{\leq\eta}$ is p-stable and derive a contradiction. To this end, choose $a \in \mathbb{R}_+$ such that $0 < a < \frac{\eta}{p+1}$ and let $c = \{a\}$. Then, we have $c^i = \{i \cdot a\}$ for every $i \geq 1$. Recall from Example 2.10 that $\mathbf{1}_{\leq\eta} = \{0\}$. Hence, $c^{(p)} = \mathbf{1}_{\leq\eta} + c^1 + c^2 + \cdots + c^p = \{0, a, 2a, \ldots, pa\}$, whereas $c^{(p+1)} = \{0, a, 2a, \ldots, pa, (p+1)a\}$. Indeed, $(p+1)a \in c^{(p+1)}$, because $\min(c^{(p+1)}) = 0$ and $(p+1)a < \eta$ due to $a < \frac{\eta}{p+1}$. Hence, $c^{(p)} \neq c^{(p+1)}$, which contradicts the above assumption that $\mathrm{Trop}^+_{\leq\eta}$ is p-stable. □

5.2 Convergence in Non-uniformly Stable Semirings

Let S be a naturally ordered semiring. Consider a vector-valued function $\mathbf{f}$ with component functions $(f_1, \dots, f_N)$ over variables $\mathbf{x} = (x_1, \dots, x_N)$. In particular, $\mathbf{f}$ is a map $\mathbf{f} : S^N \to S^N$. Recall that $\mathbf{f}$ is said to be *stable* if there exists $q \geq 0$ such that $\mathbf{f}^{(q)}(\mathbf{0}) = \mathbf{f}^{(q+1)}(\mathbf{0})$. If $\mathbf{f}$ was the ICO of a recursion, then being q-stable means the naïve algorithm terminates in q steps. In this section, we prove a result (Theorem 5.10) that essentially says, if the semiring S is stable, and if the ICO $\mathbf{f}$ is a vector-valued multi-variate polynomial function in S, then the naïve algorithm converges in a finite number of steps. We take the opportunity to develop some terminologies and techniques to prove stronger results in the sections that follow.

The following example illustrates how a univariate quadratic function can be shown to be stable in a stable semiring.[6]

Example 5.5. Consider a single polynomial, i.e., $N = 1$:

$$f(x) = b + ax^2. \tag{32}$$

Then:

$$\begin{aligned}
f^{(0)}(0) &= 0 \\
f^{(1)}(0) &= b \\
f^{(2)}(0) &= b + ab^2 \\
f^{(3)}(0) &= b + a(b + ab^2)^2 = b + ab^2 + 2a^2b^3 + a^3b^4 \\
f^{(4)}(0) &= b + a(b + ab^2 + 2a^2b^3 + a^3b^4)^2 = b + ab^2 + 2a^2b^3 + 5a^3b^4 + \cdots .
\end{aligned}$$

(Note that we overloaded notations in the above to make the expressions simpler to parse. We wrote $2ab^2$ to mean $a \otimes b \otimes b \oplus a \otimes b \otimes b$. In particular, integer coefficients are used to denote repeated summations in the semiring, whose $\oplus$ and $\otimes$ operators are already simplified to $+$ and $\cdot$.) It can be shown by induction that, when $q \geq n$, the coefficient of $a^n b^{n+1}$ is "stabilized" and it is the Catalan number $\frac{1}{n+1}\binom{2n}{n}$:

$$f^{(q)}(0) = \sum_{n=0}^{q} \frac{1}{n+1}\binom{2n}{n} a^n b^{n+1} + \sum_{n>q} \lambda_n^{(q)} a^n b^{n+1}. \tag{33}$$

The coefficients $\lambda_n^{(q)} \in \mathbb{N}$ for $n > q$ may not be in their "final form" yet. Now, suppose the element $c := ab$ is q-stable. Then, from identity (33), we can show that f is also q-stable. To see this, note that if c is q-stable, then $c^{(m)} = c^{(q)}$ for any $m > q$ (recall the notation defined in Equation (30)). Thus, for any $m > q$, a term $a^m b^{m+1}$ in the expansion of $f^{(q)}(0)$ will be "absorbed" by the earlier terms:

$$\sum_{n=0}^{q} a^n b^{n+1} + a^m b^{m+1} = b\left(c^{(q)} + c^m\right) = b\left(c^{(m-1)} + c^m\right) = bc^{(m)} = bc^{(q)} = \sum_{n=0}^{q} a^n b^{n+1}. \tag{34}$$

A simple way to see how the Catalan number shows up is to let $q \to \infty$ in the formal power series sense [76]. In this case, $f^{(\omega)}(0) = b + a[f^{(\omega)}(0)]^2$, which is then solved directly by Newton's generalized binomial expansion:

$$f^{(\omega)}(0) = \frac{1}{2a}\left(1 - \sqrt{1-4ab}\right) = \frac{1}{2a}\left(1 - \sum_{n=0}^{\infty}\binom{1/2}{n}(-1)^n a^n b^n\right) = \sum_{n=0}^{\infty} \frac{1}{n+1}\binom{2n}{n} a^n b^{n+1}. \tag{35}$$

[6] A similar example can be found in Reference [33], where the concept of *quasi square root* is introduced. However, the formula for the Catalan number cited in the book is incorrect.

Our proof of Theorem 1.2 is based on the two observations in this example. First, we prove in this section that there exists a finite set of monomials, like ab here, whose stability implies the stability of f. In addition, we prove in the next section that the coefficients, like $\lambda_n^{(q)}$ above, reach quickly their final values. Notice that, in general, no closed form like Equation (35) exists for $f^{(\omega)}$; for example, if f has degrees ≥ 5, then we cannot hope to always obtain such closed-form formulas.

We next introduce several notations that we also need in the next section. Given a positive integer k, a tuple $\mathbf{z} = (z_1, \dots, z_k)$ of symbols or variables, and a tuple $\mathbf{v} = (v_1, \dots, v_k) \in \mathbb{N}^k$ of non-negative integers, we denote

$$\mathbf{z}^{\mathbf{v}} := \prod_{i=1}^{k} z_i^{v_i}, \tag{36}$$

where the product is the product operator of the semiring under consideration. With this notation, the ith component function of a vector-valued polynomial function $\mathbf{f} = (f_1, \dots, f_N)$ can be written in the following form:

$$f_i(\mathbf{x}) = \sum_{\mathbf{v} \in V_i} a_{i,\mathbf{v}} \cdot \mathbf{x}^{\mathbf{v}}, \tag{37}$$

where the $a_{i,\mathbf{v}}$ are constants in the semiring's domain, and V_i is a set of length-N vectors of non-negative integers.

We are interested in the formal expansion of $\mathbf{f}^{(q)}(0)$ in the same way we expressed iterative applications of f in Example 5.5. The device to express these expansions formally is **context-free languages (CFL)**, as was done in a long history of work in automata theory and formal languages [19, 41].

The non-terminals of the grammar for our CFL consist of all variables (in $\mathbf{x}$) occurring in $\mathbf{f}$. Every constant $a_{i,\mathbf{v}}$ (shown in Reference (37)) corresponds to a distinct terminal symbol in the grammar. For example, even if $a_{i,\mathbf{v}} = a_{i',\mathbf{v}'}$ for $(i, \mathbf{v}) \neq (i'\mathbf{v}')$, we will consider them different symbols in the symbol set Γ of the CFL. Note also that every monomial $\mathbf{x}^{\mathbf{v}}$ has a corresponding (symbolic) coefficient $a_{i,\mathbf{v}}$, even if the coefficient is 1. For example, $1 + x^2y$ becomes $a + bx^2y$, with coefficients $a = 1$ and $b = 1$. The production rules are constructed from Reference (37) as follows: For every monomial $a_{i,\mathbf{v}}\mathbf{x}^{\mathbf{v}}$, with $\mathbf{v} = (v_1, \dots, v_N) \in V_i$, there is a rule:

$$x_i \longrightarrow a_{i,\mathbf{v}} \underbrace{x_1 \cdots x_1}_{v_1 \text{ times}} \underbrace{x_2 \cdots x_2}_{v_2 \text{ times}} \dots \underbrace{x_N \cdots x_N}_{v_N \text{ times}}. \tag{38}$$

Given a parse tree T (from the grammar above), define the *yield* $Y(T)$ of T to be the product of all terminal symbols at the leaves of T. For $i \in [N]$ and a positive integer q, let $\mathcal{T}_q^i$ denote the set of all parse trees of the above grammar, with starting symbol x_i and depth $\leq q$. The following simple but fundamental fact was observed in Reference [19] (for completeness, we include an inductive proof of this fact in the appendix):

LEMMA 5.6 ([19]). *Given integers $q \geq 0$ and $i \in [N]$, the ith component of the vector $\mathbf{f}^{(q)}(\mathbf{0})$, denoted by $(\mathbf{f}^{(q)}(0))_i$, can be expressed in terms of the yields of short parse-trees:*

$$(\mathbf{f}^{(q)}(0))_i = \sum_{T \in \mathcal{T}_q^i} Y(T). \tag{39}$$

Example 5.7. Consider the following map: $\mathbf{f} = (f_1, f_2)$:

$$\begin{bmatrix} X \\ Y \end{bmatrix} \rightarrow \begin{bmatrix} aXY + bY + c \\ uXY + vX + w \end{bmatrix}. \tag{40}$$

$$X \to aXY \mid bY \mid c \qquad\qquad Y \to uXY \mid vX \mid w$$

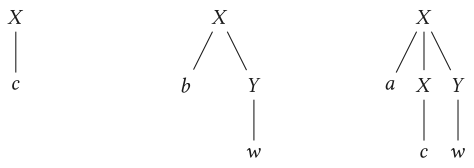


Fig. 3. X-Parse trees of depth ≤ 2 for the grammar in Example 5.7.

Applying the map twice, we obtain

$$\begin{bmatrix} X \\ Y \end{bmatrix} \to \begin{bmatrix} aXY + bY + c \\ uXY + vX + w \end{bmatrix} \to \begin{bmatrix} a(aXY + bY + c)\,(uXY + vX + w) + b(uXY + vX + w) + c \\ u(aXY + bY + c)\,(uXY + vX + w) + v(aXY + bY + c) + w \end{bmatrix}.$$

This means the first component of $\mathbf{f}^{(1)}(0)$ is $(\mathbf{f}^{(1)}(0))_1 = c$ and the first component of $\mathbf{f}^{(2)}(0)$ is $(\mathbf{f}^{(2)}(0))_1 = acw + bw + c$. The same result can be obtained via summing up the yields of parse trees of depth ≤ 1 or ≤ 2, respectively, of the following grammar, as shown in Figure 3.

The fact that the symbolic expansions of $\mathbf{f}^{(q)}$ can be expressed in terms of a CFL gives us another advantage: We can make use of Parikh's theorem [65] to further characterize the terms $Y(T)$ in expression (39). Note that the yield $Y(T)$ can be thought of as a word w in the CFL, where we "concatenate" (i.e., multiply) all terminal symbols in T from left to right. Since our multiplicative operator is commutative, only the multiplicities of the symbols matter in differentiating two yields $Y(T)$ and $Y(T')$. The multiplicities of the symbols are formalized by the *Parikh image* of the word w.

More formally, consider our alphabet Σ containing all terminal symbols in the CFL. By renaming, we can assume $\Sigma = \{a_1, \dots, a_M\}$, and we write $\mathbf{a} = (a_1, \dots, a_M)$ to denote the vector of all terminal symbols. The *Parikh image* of a word $w \in \Sigma^*$, denoted by $\Pi(w)$, is the vector $\Pi(w) = (k_1, \dots, k_M) \in \mathbb{N}^M$, where k_i is the number of occurrences of a_i in w. With this notation, Equation (39) can be expressed as:

$$(\mathbf{f}^{(q)}(\mathbf{0}))_i = \sum_{T \in \mathcal{T}_q^i} Y(T) = \sum_{T \in \mathcal{T}_q^i} \mathbf{a}^{\Pi(Y(T))}. \tag{41}$$

Parikh's theorem states that the signatures $\Pi(Y(T))$ in the above expressions have a particular format. To state Parikh's theorem, we need the notion of "semi-linear sets."

Definition 5.8 (Semi-linear Sets). Given an integer $M > 0$, a set $\mathcal{L} \subseteq \mathbb{N}^M$ is said to be *linear* if there exist vectors $\mathbf{v}_0, \mathbf{v}_1, \dots, \mathbf{v}_\ell \in \mathbb{N}^M$ such that:

$$\mathcal{L} = \{\mathbf{v}_0 + k_1\mathbf{v}_1 + \cdots + k_\ell \mathbf{v}_\ell \mid k_1, \dots, k_\ell \in \mathbb{N}\}.$$

A set $\mathcal{L} \subseteq \mathbb{N}^M$ is called *semi-linear* if it is a finite union of linear sets.

Theorem 5.9 (Parikh's Theorem [65]). *Let G be a context-free grammar with terminal symbols Σ, and let $L(G) \subseteq \Sigma^*$ be the language generated by G. Then, $\Pi(L(G)) \subseteq \mathbb{N}^M$ is a semi-linear set.*

We now have all the tools to prove the main theorem of this section:

Theorem 5.10. *If the semiring $\mathbf{S}$ is stable, then every polynomial function $\mathbf{f} : \mathbf{S}^N \to \mathbf{S}^N$ is stable.*

Proof. From Parikh's theorem and identity (41), there is a *finite* collection C, where every member of C is a tuple of vectors $(\mathbf{v}_0, \mathbf{v}_1, \dots, \mathbf{v}_\ell) \in (\mathbb{N}^M)^{\ell+1}$ (tuples in C may have different lengths $\ell + 1$), satisfying the following condition. For every parse-tree $T \in \mathcal{T}_q^i$, there exists

$(\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_\ell) \in C$ and a coefficient vector $(k_1, \ldots, k_\ell) \in \mathbb{N}^\ell$ for which

$$Y(T) = \mathbf{a}^{\Pi(Y(T))} = \mathbf{a}^{\mathbf{v}_0} \prod_{i=1}^{\ell} (\mathbf{a}^{\mathbf{v}_i})^{k_i}. \tag{42}$$

Conversely, for every tuple $(\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_\ell) \in C$ and every coefficient vector $(k_1, \ldots, k_\ell)$ there exists a parse-tree T in some $\mathcal{T}_q^i$ for which the identity holds.

Our proof strategy is as follows: We shall define below a finite set $B \subseteq \mathbb{N}^M$ of exponent vectors and express $f_i^{(q)}(0)$ by grouping the terms $\mathbf{a}^{\mathbf{v}}$ in Equation (41) that have the same exponent vector $\mathbf{v} \in \mathbb{N}^M$:

$$(\mathbf{f}^{(q)}(\mathbf{0}))_i = \sum_{\mathbf{v} \in B} \lambda_{\mathbf{v}}^{(q)} \mathbf{a}^{\mathbf{v}} + \sum_{\mathbf{w} \notin B} \lambda_{\mathbf{w}}^{(q)} \mathbf{a}^{\mathbf{w}}, \tag{43}$$

where the coefficients $\lambda_{\mathbf{v}}^{(q)}$ are integers that may change over time as q increases:

$$\lambda_{\mathbf{v}}^{(q)} = |\{T \in \mathcal{T}_q^i : \Pi(Y(T)) = \mathbf{v}\}|. \tag{44}$$

Here, for an element $s \in S$ and a positive integer $\lambda \in \mathbb{N}$, we write λs to mean $s + s + \cdots + s$, λ times. In particular, this multiplication of λ and s should not be confused with the multiplications of symbols in $\mathbf{a}$ which are done over the multiplicative operator of the semiring. We then prove two claims. For *sufficiently large* q, the following hold:

- **Claim 1** the coefficients $\lambda_{\mathbf{v}}^{(q)} \in \mathbb{N}$ for $\mathbf{v} \in B$ no longer change (they "converge")
- **Claim 2** for every $\mathbf{w} \notin B$, we have

$$\sum_{\mathbf{v} \in B} \lambda_{\mathbf{v}}^{(q)} \mathbf{a}^{\mathbf{v}} + \mathbf{a}^{\mathbf{w}} = \sum_{\mathbf{v} \in B} \lambda_{\mathbf{v}}^{(q)} \mathbf{a}^{\mathbf{v}}. \tag{45}$$

These two claims together show that $\mathbf{f}$ is stable: $(\mathbf{f}^{(q)}(\mathbf{0}))_i = \sum_{\mathbf{v} \in B} \lambda_{\mathbf{v}}^{(q)} \mathbf{a}^{\mathbf{v}}$ for large q.

We start by defining the set B. Fix a sufficiently large integer p to be defined later. Define B to be[7]:

$$B := \{\mathbf{v}_0 + k_1\mathbf{v}_1 + \cdots + k_\ell\mathbf{v}_\ell \mid (\mathbf{v}_0, \ldots, \mathbf{v}_\ell) \in C \text{ and } \mathbf{k} = (k_1, \ldots, k_\ell) \in \mathbb{N}^\ell \text{ where } \|\mathbf{k}\|_\infty \leq p\}. \tag{46}$$

To prove Claim 1, note that, if the parse tree T has depth h, then $\|\Pi(Y(T))\|_1 \geq h$. Consequently, all parse trees whose depths are strictly greater than $\max_{\mathbf{v} \in B} \|\mathbf{v}\|_1$ can no longer contribute to increasing $\lambda_{\mathbf{v}}^{(q)}$ with $\mathbf{v} \in B$; thus, these coefficients converge after iteration number $\max_{\mathbf{v} \in B} \|\mathbf{v}\|_1$. This is a finite number, because (46) implies:

$$\max_{\mathbf{v} \in B} \|\mathbf{v}\|_1 \leq \max_{(\mathbf{v}_0, \ldots, \mathbf{v}_\ell) \in C} (1 + p \cdot \ell) \max_{i \in \{0, \ldots, \ell\}} \|\mathbf{v}_i\|_1. \tag{47}$$

Claim 2 is proved as follows: Given $\mathbf{w} \notin B$, as mentioned above, from Parikh's theorem, we know there exists $(\mathbf{v}_0, \ldots, \mathbf{v}_\ell) \in C$ and $\mathbf{k} \in \mathbb{N}^\ell$ (but $\|\mathbf{k}\|_\infty > p$) such that $\mathbf{w} = \mathbf{v}_0 + k_1\mathbf{v}_1 + \cdots + k_\ell\mathbf{v}_\ell$. Rewrite the monomial $\mathbf{a}^{\mathbf{w}}$ from the "basis" $(\mathbf{v}_0, \ldots, v_\ell)$ by:

$$\mathbf{a}^{\mathbf{w}} = \mathbf{a}^{\mathbf{v}_0} (\mathbf{a}^{\mathbf{v}_1})^{k_1} \cdots (\mathbf{a}^{\mathbf{v}_\ell})^{k_\ell} = m_0 m_1^{k_1} \cdots m_\ell^{k_\ell},$$

where, to simplify notations, we define the monomials $m_i := \mathbf{a}^{\mathbf{v}_i}$.

[7]Given a vector $\mathbf{x} = (x_1, \ldots, x_d) \in \mathbb{N}^d$ for some constant d, we use $\|\mathbf{x}\|_\infty$ to denote $\max_{i \in [d]} x_i$ and $\|\mathbf{x}\|_1$ to denote $\sum_{i \in [d]} x_i$.

First, let us assume for simplicity that only one of the k_i is more than p. Without loss of generality, assume $k_\ell > p$. Then, as long as p is at least the stability index of m_ℓ, thanks to (31), we have

$$\begin{aligned}\sum_{i=0}^{p} m_0 m_1^{k_1} \cdots m_{\ell-1}^{k_{\ell-1}} m_\ell^i + m_0 m_1^{k_1} \cdots m_\ell^{k_\ell} &= m_0 m_1^{k_1} \cdots m_{\ell-1}^{k_{\ell-1}} m_\ell^{(p)} + m_0 m_1^{k_1} \cdots m_\ell^{k_\ell} \\ &= m_0 m_1^{k_1} \cdots m_{\ell-1}^{k_{\ell-1}} m_\ell^{(k_\ell-1)} + m_0 m_1^{k_1} \cdots m_\ell^{k_\ell} \\ &= m_0 m_1^{k_1} \cdots m_{\ell-1}^{k_{\ell-1}} m_\ell^{(k_\ell)} \\ &= m_0 m_1^{k_1} \cdots m_{\ell-1}^{k_{\ell-1}} m_\ell^{(p)} \\ &= \sum_{i=0}^{p} m_0 m_1^{k_1} \cdots m_{\ell-1}^{k_{\ell-1}} m_\ell^i.\end{aligned}$$

In particular, the term $\mathbf{a}^{\mathbf{w}}$ is "absorbed" by the sum of terms of the form $\mathbf{a}^{\mathbf{v}} = m_0 m_1^{k_1} \cdots m_{\ell-1}^{k_{\ell-1}} m_\ell^i$ for $i \leq p$. Since all such $\mathbf{v}$ are in B (because $k_i \leq p$ for all $i < \ell$), we have just proved Equation (45) for this simple case of $\mathbf{w}$.

Second, when more than one of the k_i is greater than p, we can w.l.o.g. assume $k_\ell > p$. The above reasoning shows that the term $\mathbf{a}^{\mathbf{w}}$ is "absorbed" by the sum of terms $\mathbf{a}^{\mathbf{v}}$, where $\mathbf{v}$ has one less $k_i > p$. By induction, it follows that all of them will be absorbed by the sum $\sum_{\mathbf{v} \in B} \mathbf{a}^{\mathbf{v}}$. □

5.3 Convergence in Uniformly Stable Semirings

We consider the next case, when the semiring $\boldsymbol{S}$ is uniformly stable; in other words, there exists $p \geq 0$ such that every element in $\boldsymbol{S}$ is p-stable. In this case, we can strengthen Theorem 5.10 and prove a tighter upper bound on the number of steps needed for convergence. We say that the polynomial function $\mathbf{f} : S^N \to S^N$ is *linear* if every monomial (8) has total degree ≤ 1. The main theorem (Theorem 5.12) is proved via proving a 1-dimensional version of it, which is then generalized to N dimensions by applying Theorem 3.4. A univariate polynomial is of the following form:

$$f(x) \stackrel{\text{def}}{=} a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n. \tag{48}$$

The following lemma generalizes special cases studied by Gondran [32, 33]:

Lemma 5.11. *Let $\boldsymbol{S}$ be a p-stable semiring, and let f be a univariate polynomial (48). Then: (a) If $p = 0$, then f is 1-stable; (b) If f is linear, then it is $p + 1$-stable; (c) In general, f is $p + 2$-stable.*

Proving the lemma will be the bulk of work in this section. Before doing so, let us state and prove the main theorem of the section, concerning the N-dimensional case.

Theorem 5.12. *Assume that the semiring $\boldsymbol{S}$ is p-stable, and let $\mathbf{f} : S^N \to S^N$ be a polynomial function. Then:*

(1) The function $\mathbf{f}$ is $\sum_{i=1}^{N}(p+2)^i$-stable; if $\mathbf{f}$ is linear, then it is $\sum_{i=1}^{N}(p+1)^i$-stable.
(2) If $p = 0$, then the function $\mathbf{f}$ is N-stable.

Proof. To prove item (1), consider the c-clone of polynomial functions. By Lemma 5.11, each univariate polynomial is $p + 2$-stable, and therefore Theorem 3.4 implies that every polynomial function $S^N \to S^N$ is $\sum_{i=1,N}(p+2)^i$-stable. If $\mathbf{f}$ is linear, then we consider the c-clone of linear functions and derive similarly that $\mathbf{f}$ is $\sum_{i=1,N}(p+1)^i$-stable.

For item (2), when $p = 0$, consider the expansion of $(\mathbf{f}^{(q)}(\mathbf{0}))_i$ shown in Equation (41). We will prove that, when $q > N$, all parse trees $T \in \mathcal{T}_q^i - \mathcal{T}_N^i$ are "absorbed" by those in $\mathcal{T}_N^i$:

$$(\mathbf{f}^{(q)}(\mathbf{0}))_i = \sum_{T \in \mathcal{T}_q^i} Y(T) = \sum_{T \in \mathcal{T}_N^i} Y(T) = (\mathbf{f}^{(N)}(\mathbf{0}))_i \qquad \text{when } q \geq N. \tag{49}$$

To see this, consider a parse tree $T \in \mathcal{T}_q^i - \mathcal{T}_N^i$. Since its depth is $> N$, there is a path from the root to a leaf containing a repeated variable symbol, say, x_j for some $j \in [N]$. Let w be the word that the lower copy of x_j derives, then the higher copy of x_j derives some word of the form uwv, and the tree derives the word $Y(T) = auwvb$ for some words a, b. Let T' be the tree obtained from T by replacing the derivation of the higher-copy of x_j with the derivation of the lower copy of x_j. Then, $Y(T') = awb$. Now, since $p = 0$, we have $Y(T') + Y(T) = awb + auwvb = awb(1 + uv) = awb = Y(T')$. Repeating this process, by induction, it follows that $Y(T)$ is absorbed by $\sum_{T \in \mathcal{T}_N^i} Y(T)$. □

In the rest of this section, we prove the main Lemma 5.11.

Proof of Lemma 5.11. To prove (a), we recall that in a 0-stable semiring, $1 + c = 1$ for every $c \in S$. If f is the polynomial in Equation (48), then we have $f^{(1)}(0) = a_0$, and f is 1-stable because

$$f^{(2)}(0) = a_0 + \sum_{i=1}^{n} a_0^i a_i = a_0 \left(1 + \sum_{i=1,n} a_0^{i-1} a_i\right) = a_0 1 = a_0 = f^{(1)}(0).$$

To prove (b), note that linearity means $f(x) = a_0 + a_1 x$, and $f^{(n+1)}(0) = a_0 + a_1 a_0 + a_1^2 a_0 + \cdots + a_1^n a_0 = a_1^{(n)} a_0$, which implies $f^{(p+1)}(0) = f^{(p+2)}(0)$, since a_1 is p-stable.

Part (c) is the most interesting result. Our strategy follows that of the proof of Theorem 5.10. We start by writing an expansion of f as was done in Equation (41) and the regrouping (43):

$$f^{(q)}(0) = \sum_{T \in \mathcal{T}_q} Y(T) = \sum_{T \in \mathcal{T}_q} \mathbf{a}^{\Pi(Y(T))} = \sum_{\mathbf{v} \in B} \lambda_{\mathbf{v}}^{(q)} \mathbf{a}^{\mathbf{v}} + \sum_{\mathbf{w} \notin B} \lambda_{\mathbf{w}}^{(q)} \mathbf{a}^{\mathbf{w}}. \tag{50}$$

The main difference is that in the CFL for the function (48), the terminal set is $\Sigma = \{a_0, \ldots, a_n\}$; and there is only one variable, so we remove the index i from Equation (41): $\mathcal{T}_q$ is the set of all parse trees of depth at most q, whose root is x.

The exponent set B is defined in exactly the same way as that in (46). However, taking advantage of the fact that f in Equation (48) is univariate, we show in Proposition 5.13 below that the collection C has only *one* specific tuple of vectors $(\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_n)$ (defined in (52)). In particular,

$$B := \{\mathbf{v}_0 + k_1\mathbf{v}_1 + \cdots + k_n\mathbf{v}_n \mid \mathbf{k} = (k_1, \ldots, k_n) \in \mathbb{N}^n \text{ where } \|\mathbf{k}\|_\infty \leq p\}. \tag{51}$$

Finally, to show that f has stability $p + 2$, we prove in Proposition 5.14 below that the sum $\sum_{\mathbf{v} \in B} \lambda_{\mathbf{v}}^{(q)} \mathbf{a}^{\mathbf{v}}$ is "finalized" (unchanged) when $q \geq p + 2$. The fact that the remaining coefficients $\lambda_{\mathbf{w}}^{(q)}$, $\mathbf{w} \notin B$, are absorbed by the sum over B was shown in Equation (45). The lemma is thus proved, modulo the proofs of the promised propositions below. □

The following proposition is a specialization of Parikh's theorem to the particular grammar arising from the polynomial (48):

Proposition 5.13. *Define the following vectors in* $\mathbb{N}^{n+1}$*:*

$$\mathbf{v}_0 = (1, 0, \ldots, 0) \qquad \mathbf{v}_i = (i-1, 0, \ldots, 0, 1, 0, \ldots 0) \qquad i \in [n]. \tag{52}$$

Let $\mathcal{T}$ be the set of all *parse trees for the CFG defined for the polynomial (48), then the Parikh's images of their yields can be characterized precisely by:*

$$\{\Pi(Y(T)) \mid T \in \mathcal{T}\} = \{\mathbf{v}_0 + k_1\mathbf{v}_1 + \cdots + k_n\mathbf{v}_n \mid \mathbf{k} \in \mathbb{N}^n\}. \tag{53}$$

Proof. For the forward direction, let T be any parse tree of the grammar. Let k_i denote the number of derivation rules of the form $x \longrightarrow a_i x \cdots x$ (x occurs i times on the RHS) that were used in the tree. Since each such rule produces one copy of a_i in the yield $Y(T)$, it follows that $\Pi(Y(T)) = (k_0, k_1, \dots, k_n)$. The numbers k_i are related to one another tightly. Consider the sub-tree of T containing only the internal nodes (i.e., nodes corresponding to x, not the terminal symbols), then this sub-tree has exactly $k_0 + \sum_{i=1}^n k_i$ nodes and $\sum_{i=1}^n ik_i$ edges. In a tree, the number of edges is 1 less than the number of nodes. Hence, $k_0 = 1 + \sum_{i=1}^n (i-1)k_i$, which implies $\Pi(Y(T)) = \mathbf{v}_0 + \sum_{i=1}^n k_i \mathbf{v}_i$.

For the backward direction, consider an arbitrary vector $\mathbf{k} = (k_1, \dots, k_n) \in \mathbb{N}^n$. We show that we can construct a parse-tree T whose yield satisfies $\Pi(Y(T)) = \mathbf{v}_0 + \sum_{i=1}^n k_i \mathbf{v}_i$. We prove this by induction on $\|\mathbf{k}\|_1$. The base case when $\|\mathbf{k}\|_1 = 0$ is trivial: The parse tree represents a single rule $x \longrightarrow a_0$. Consider $\|\mathbf{k}\|_1 > 0$, and assume $k_i > 0$. Let $\mathbf{k}' = (k'_1, \dots, k'_n)$ be the vector defined by $k'_j = k_j$ when $j \neq i$, and $k'_i = k_i - 1$. Then, by the induction hypothesis, there is a parse tree T' where $\Pi(Y(T')) = \mathbf{v}_0 + \sum_{j=1}^n k'_j \mathbf{v}_j$. The parse-tree T' must have used at least one rule of the form $x \longrightarrow a_0$ (otherwise, the tree is infinite). Construct the tree T from T' by replacing the rule $x \longrightarrow a_0$ with $x \longrightarrow a_i x \cdots x$; and then each of the i new x-nodes derives a_0. Then, it is trivial to verify that $\Pi(Y(T)) = \mathbf{v}_0 + \sum_{j=1}^n k_j \mathbf{v}_j$. □

Proposition 5.14. *After iteration $q \geq p+2$, the sum $\sum_{\mathbf{v} \in B} \lambda_{\mathbf{v}}^{(q)} \mathbf{a}^{\mathbf{v}}$ in Equation (50) remains unchanged.*

Proof. Partition B into two sets $B = B_1 \cup B_2$:

$$B_1 := \{\mathbf{v}_0 + k_1\mathbf{v}_1 + \cdots + k_n\mathbf{v}_n \mid \mathbf{k} = (k_1, \dots, k_n) \in \mathbb{N}^n \text{ where } \|\mathbf{k}\|_\infty \leq p \text{ and } \|\mathbf{k}\|_1 \leq p\}, \quad (54)$$

$$B_2 := \{\mathbf{v}_0 + k_1\mathbf{v}_1 + \cdots + k_n\mathbf{v}_n \mid \mathbf{k} = (k_1, \dots, k_n) \in \mathbb{N}^n \text{ where } \|\mathbf{k}\|_\infty \leq p \text{ and } \|\mathbf{k}\|_1 > p\}. \quad (55)$$

Note that the condition $\|\mathbf{k}\|_\infty \leq p$ in the definition of B_1 is redundant (because $\|\mathbf{k}\|_1 \leq p$ implies $\|\mathbf{k}\|_\infty \leq p$), but we left it there to mirror precisely the definition of B in Equation (51).

Recall from Equation (44) that $\lambda_{\mathbf{v}}^{(q)}$ is the number of parse trees $T \in \mathcal{T}_q$ where $\Pi(Y(T)) = \mathbf{v}$; furthermore, if $\mathbf{v} = \mathbf{v}_0 + \sum_{i=1}^n k_i \mathbf{v}_i$, then $\|\mathbf{k}\|_1 \geq \mathsf{depth}(T)$, because $\|\mathbf{k}\|_1$ is the number of internal nodes of the tree, which is at least its depth. Hence, when $\mathbf{v} \in B_1$, only trees of depth $\leq \|\mathbf{k}\|_1 \leq p$ can contribute to increasing $\lambda_{\mathbf{v}}^{(q)}$. In other words, these coefficients are fixed after iteration p.

It remains to show that the sum $\sum_{\mathbf{v} \in B_2} \lambda_{\mathbf{v}}^{(q)} \mathbf{a}^{\mathbf{v}}$ is fixed after iteration $q \geq p+2$. To this end, note the following: For any element s in a p-stable semiring, and any positive integer $\lambda \geq p+1$, we have

$$\lambda s := \underbrace{s + s + \cdots + s}_{\lambda \text{ times}} = (1 + 1 + \cdots + 1)s = (1 + 1 + 1^2 \cdots + 1^{\lambda-1})s = 1^{(\lambda-1)}s = 1^{(p)}s = (p+1)s.$$

In particular, the sum that we want to hold fixed can be written as

$$\sum_{\mathbf{v} \in B_2} \lambda_{\mathbf{v}}^{(q)} \mathbf{a}^{\mathbf{v}} = \sum_{\mathbf{v} \in B_2} \min\{\lambda_{\mathbf{v}}^{(q)}, p+1\} \mathbf{a}^{\mathbf{v}}. \quad (56)$$

It is thus sufficient to prove that, for any $\mathbf{v} \in B_2$, we have $\lambda_{\mathbf{v}}^{(q)} \geq p+1$ after iteration $q \geq p+2$. We will prove something a little stronger:

Claim: For any $\mathbf{v} = \mathbf{v}_0 + \sum_i k_i \mathbf{v}_i \in B_2$, we have $\lambda_{\mathbf{v}}^{(\|\mathbf{k}\|_\infty + 2)} \geq \|\mathbf{k}\|_1$; namely, there are at least $\|\mathbf{k}\|_1$ many parse trees T with depth at most $2 + \|\mathbf{k}\|_\infty$ and whose Parikh's image is $\Pi(Y(T)) = \mathbf{v}$.

The claim implies what we desire, because for any $\mathbf{v} \in B_2$, $\lambda_{\mathbf{v}}^{(p+2)} \geq \lambda_{\mathbf{v}}^{(\|\mathbf{k}\|_\infty + 2)} \geq \|\mathbf{k}\|_1 \geq p+1$.

We prove the claim by induction on $\|\mathbf{k}\|_\infty$. Consider the base case when $\|\mathbf{k}\|_\infty = 1$. Let $J := \{j \in [n] \mid k_j = 1\}$ and $i = \max\{j \mid j \in J\}$. We construct parse trees by stitching together the rules (i.e., sub-trees) $x \longrightarrow a_j x \dots x$ for $j \in J$ and fill up the leaves with $x \longrightarrow a_0$ rules. At the root of the tree, we will apply $x \longrightarrow a_i x \dots x$ for maximum flexibility, where there are i x-nodes to fit the other $|J| - 1$ rules in. Clearly, there are $\binom{i}{|J|-1}((|J|-1)!) \geq |J| \geq \|\mathbf{k}\|_1$ distinct trees that can be constructed this way.

For the inductive step, assume $\|\mathbf{k}\|_\infty > 1$. Similar to the base case, define $J := \{j \in [n] \mid k_j = \|\mathbf{k}\|_\infty\}$ and $i = \max\{j \mid j \in J\}$. Let $\mathbf{k}'$ be the vector obtained from $\mathbf{k}$ by lowering each k_j by 1, for $j \in J$; namely, define $k'_j = \min\{k_j, \|\mathbf{k}\|_\infty - 1\}$. Then, by the induction hypothesis, there are at least $\|\mathbf{k}'\|_1$ parse trees T' for which $\Pi(Y(T')) = \mathbf{v}_0 + \sum_\ell k'_\ell \mathbf{v}_\ell$. From each of these T', we construct our tree T by stitching together T' and the rules (i.e., sub-trees) $x \longrightarrow a_j x \dots x$, $j \in J$. (Implicitly, x-leaves are filled up with $x \longrightarrow a_0$ rules.) For the root of T, we use the rule $x \longrightarrow a_i x \dots x$. There are i sub-tree slots to fill, where we can use $|J| - 1$ sub-trees of the form $x \longrightarrow a_j x \dots x$, $j \in J - \{i\}$, and T', for a total of $|J|$ subtrees. (Note that $|J| \leq i$, and thus we have more slots than sub-trees.)

If $i \geq 2$, then the total number of trees that can be obtained this way is[8]

$$\|\mathbf{k}'\|_1 \binom{i}{|J|}(|J|)! \geq (\|\mathbf{k}\|_1 - |J|)i \geq (\|\mathbf{k}\|_1 - i)i \geq \|\mathbf{k}\|_1 \qquad \text{as desired.} \tag{57}$$

If $i = 1$, then we have to do a bit of extra work. The total number of trees T constructed above is only $\|\mathbf{k}'\|_1 = \|\mathbf{k}\|_1 - 1$, because the root $x \longrightarrow a_1 x$ has only one slot to fill T' in. Let us refer to this set as U_1. We construct another set U_2 of trees by letting T' be at the top and find a slot to fit the sub-tree $x \longrightarrow a_1 x$ in.

We refer to a node $x \longrightarrow a_\ell x \dots x$ of a parse-tree as an ℓ-node. Since $\|\mathbf{k}\|_\infty \leq p$ and $\|\mathbf{k}\|_1 > p$, there must be some $\ell > 1$ for which $k_\ell > 0$. In particular, there must be a sub-tree of T' whose root node is an ℓ-node with $\ell > 1$. In T', find the left-most such ℓ-node, and the left-most 0-node ($x \longrightarrow a_0$) below it. Replace the $x \longrightarrow a_0$ rule with $x \longrightarrow a_1 x \longrightarrow a_1 a_0$ sub-tree. This way of construction gives us the second set U_2 of trees T for which $\Pi(Y(T)) = \mathbf{v}$. We know $|U_1| = |U_2| \geq \|\mathbf{k}\|_1 - 1$. Furthermore, there must be at least one tree in U_1 that is not in U_2, because the tree T in U_1 with the *longest* path containing consecutive 1-nodes from the root is not in U_2. Thus, $|U_1 \cup U_2| \geq \|\mathbf{k}\|_1$, and the proof is complete. $\square$

Example 5.15. Assume a semiring that is 1-stable, and let $f(x) = a_0 + a_2x^2 + a_3x^3 + a_4x^4$. Then:

$$f^{(0)}(0) = 0 \qquad f^{(1)}(0) = a_0 \qquad f^{(2)}(0) = a_0 + a_0^2 a_2 + a_0^3 a_3 + a_0^4 a_4.$$

Next, $f^{(3)}(0)$ is a longer expression that includes monomials such as $a_0^4 a_2 a_3$ and $a_0^7 a_2 a_3 a_4$. None of these monomials appears in $f^{(2)}$, hence, the stability index for f is at least 3. However, $f^{(4)}(0) = f^{(3)}(0)$, because in any new monomial in $f^{(4)}(0)$, at least one of a_2, a_3, or a_4 has degree ≥ 2 and is absorbed by other monomials already present in $f^{(3)}(0)$: For example, $f^{(4)}(0)$ contains the new monomial $a_0^5 a_2^2 a_3$, which is absorbed, because $f^{(4)}(0)$ also contains the monomials $a_0^3 a_3$ and $a_0^4 a_2 a_3$ (they were already present in $f^{(3)}(0)$), and the following identity holds in the 1-stable semiring: $a_0^3 a_3 + a_0^4 a_2 a_3 + a_0^5 a_2^2 a_3 = a_0^3 a_3(1 + (a_0 a_2) + (a_0 a_2)^2) = a_0^3 a_3(1 + (a_0 a_2)) = a_0^3 a_3 + a_0^4 a_2 a_3$.

5.4 Convergence in Stable POPS

We will now generalize the convergence theorems from semirings to POPS. Let $\boldsymbol{P}$ be a POPS, and recall that we assume throughout this article that multiplication is strict, $x \cdot \bot = \bot$. Recall that

[8]Note that T' is distinct from any of the trees $x \longrightarrow a_j x \dots x$ for $j \in J - i$, because it has a different signature: T' contains at least two internal nodes of the form $x \longrightarrow a_\ell x \cdots x$ for $\ell > 0$ due to the fact that $\|\mathbf{k}\|_\infty < \|\mathbf{k}\|_1$.

$\boldsymbol{S} \stackrel{\text{def}}{=} \boldsymbol{P} + \bot$ is a semiring, the *core semiring* of $\boldsymbol{P}$. We call $\boldsymbol{P}$ *stable*, or *uniformly stable*, if the core $\boldsymbol{S}$ is stable or uniformly stable, respectively. We generalize now Theorems 5.10 and 5.12 from stable semirings to stable POPS.

Fix an N-tuple of polynomials $\mathbf{f} = (f_1, \ldots, f_N)$ over N variables $\mathbf{x} = (x_1, \ldots, x_N)$. We define the following directed graph $G_\mathbf{f}$: The nodes of the graph are the variables $x_1, \ldots, x_N$, and there exists an edge $x_i \longrightarrow x_j$ if the polynomial f_j depends on x_i, i.e., it contains a monomial where the factor $x_i^{\ell_i}$ has a degree $\ell_i \geq 1$. Call a variable x_i *recursive* if it belongs to a cycle or if there exists a recursive variable x_j and an edge $x_j \longrightarrow x_i$. Otherwise, we call x_i *non-recursive*.

Proposition 5.16. *If x_i is recursive, then for all $q \geq 0$, $(\mathbf{f}^{(q)}(\bot))_i \in \boldsymbol{P} + \bot$. In other words, the recursive variables cannot escape $\boldsymbol{P} + \bot$.*

Proof. We prove the statement by induction on q. When $q = 0$, the statement holds, because, at initialization, $(\mathbf{f}^{(0)}(\bot))_i = \bot$ for all i. Assume the statement holds for $q \geq 0$, and consider the value

$$(\mathbf{f}^{(q+1)}(\bot))_i = f_i((\mathbf{f}^{(q)}(\bot))_1, \ldots, (\mathbf{f}^{(q)}(\bot))_N),$$

where i is such that x_i is recursive. By assumption, f_i contains some monomial $c \cdot x_1^{\ell_1} \cdots x_N^{\ell_N}$ that includes a recursive variable x_j, and by induction hypothesis, $(\mathbf{f}^{(q)}(\bot)))_j \in \boldsymbol{P} + \bot$. Therefore, $(f^{(q+1)}(\bot))_i$ is a sum that includes the expression $c \cdot (\mathbf{f}^{(q)}(\bot))_1^{\ell_1} \cdots (\mathbf{f}^{(q)}(\bot))_N^{\ell_N}$, where the jth factor is in $\boldsymbol{P} + \bot$. Therefore, $(f^{(q+1)}(\bot))_i \in \boldsymbol{P} + \bot$, because both multiplication and addition with an element in $\boldsymbol{P} + \bot$ result in an element in $\boldsymbol{P} + \bot$, by $(u + \bot)v = uv + \bot v = uv + \bot$ and similarly $(u + \bot) + v = (u + v) + \bot$. □

Suppose the polynomial function $\mathbf{f} : \boldsymbol{P}^N \longrightarrow \boldsymbol{P}^N$ has N_0 non-recursive variables and N_1 recursive variables, where $N_0 + N_1 = N$. We write $\mathbf{f} = (\mathbf{g}, \mathbf{h})$, where $\mathbf{g} : \boldsymbol{P}^{N_1} \times \boldsymbol{P}^{N_0} \longrightarrow \boldsymbol{P}^{N_1}$ are the polynomials associated to the recursive variables, and $\mathbf{h} : \boldsymbol{P}^{N_0} \longrightarrow \boldsymbol{P}^{N_0}$ are the rest; note that $\mathbf{h}$ does not depend on any recursive variables. By Lemma 3.2, if $\mathbf{g}, \mathbf{h}$ are stable, then so is $\mathbf{f}$. We claim that the stability index of $\mathbf{h}$ is N_0. This follows easily by induction on N_0, using Lemma 3.2. If $N_0 = 1$, then $h(x) = c$ for some constant c, since it cannot depend on any variable. Assume $N_0 > 1$. Since none of the variables used by $\mathbf{h}$ is recursive, its graph $G_\mathbf{h}$ is acyclic, hence, there is one variable without an incoming edge. Write $\mathbf{h} = (\mathbf{h}', h'')$, where h'' corresponds to the variable without incoming edge, hence, it is a constant function (does not depend on any variables), and $\mathbf{h}'$ is a vector of $N_0 - 1$ non-recursive polynomials. The stability index of $\mathbf{h}'$ is $N_0 - 1$ by induction hypothesis, while that of h'' is 1, therefore, by Lemma 3.2, the stability index of $\mathbf{h}$ is $(N_0 - 1) + 1 = N_0$.

Our discussion implies:

Corollary 5.17 (Generalization of Theorem 5.10). *If the semiring $\boldsymbol{P} + \bot$ is stable, then every polynomial function $\mathbf{f} : \boldsymbol{P}^N \longrightarrow \boldsymbol{P}^N$ is stable. In particular, every* datalog$^\circ$ *program converges on the POPS $\boldsymbol{P}$.*

Corollary 5.18 (Generalization of Theorem 5.12). *Assume the POPS $\boldsymbol{P}$ is p-stable, and let $\mathbf{f} : \boldsymbol{P}^N \longrightarrow \boldsymbol{P}^N$ be a polynomial function. Then, function $\mathbf{f}$ is $\sum_{i=0}^{N}(p+2)^i$-stable; if $\mathbf{f}$ is linear, then it is $\sum_{i=0}^{N}(p+1)^i$-stable.*

In particular, every datalog$^\circ$ *program over the POPS $\boldsymbol{P}$ converges in a number of steps given by the expressions above, where N is the number of grounded IDB atoms.*

The corollaries convey a simple intuition. If the grounded datalog$^\circ$ program is recursive, then the values of the recursive atoms cannot escape the semiring $\boldsymbol{P} + \bot$; the non-recursive atoms, however, can take values outside of this semiring. This can be seen in Example 4.2: When we interpret the program over the lifted reals, $\mathbb{R}_\bot$, then the values of $T(a), T(b)$ remain $\bot$, yet those of $T(c), T(d)$ take values that are $\neq \perp$. This implies that the convergence of a recursive grounded program is tied to the stability of $\mathbf{R} + \perp$. The convergence of a non-recursive grounded program, however, follows directly from the fact that its graph is acyclic.

Finally, we prove a sufficient condition for convergence in polynomial time.

Corollary 5.19 (Generalization of the case $p = 0$ in Theorem 5.12). *Assume the POPS $\boldsymbol{P}$ is 0-stable, and let $\mathbf{f} : \boldsymbol{P}^N \to \boldsymbol{P}^N$ be a polynomial function. Then, function $\mathbf{f}$ is N-stable. In particular, every* datalog$^\circ$ *program over the POPS $\boldsymbol{P}$ converges in a number of steps that is polynomial in the size of the input database.*

The semirings $\mathbb{B}, \text{Trop}^+$ are 0-stable; the POPS $\mathbb{R}_\perp$ is also 0-stable (because $\mathbb{R}_\perp + \perp$ is the trivial semiring $\{\perp\}$, with a single element). Corollary 5.19 implies that datalog$^\circ$ converges in polynomial time on each of these semirings.

These three corollaries complete the proof of Theorem 1.2 in the introduction.

5.5 Convergence in PTIME for p-Stable Semirings when $p > 0$

Consider a datalog$^\circ$ program over a p-stable POPS $\boldsymbol{P}$. When $p = 0$, then we know that the naïve algorithm converges in polynomial time. What is its runtime when $p > 0$? Corollary 5.18 gives only an exponential upper bound, and we leave open the question whether that bound is tight. Instead, in this section, we restrict the datalog$^\circ$ program to be a linear program and ask whether it can be computed in polynomial time over a p-stable POPS. We prove two results. First, if the POPS is Trop_p^+ (which is p-stable), then the naïve algorithm converges in polynomial time. Second, for any p-stable POPS, the datalog$^\circ$ program can be computed in polynomial time by using the Floyd-Warshall-Kleene approach [52, 72]. We leave open the question whether the naïve algorithm also converges in polynomial time.

We start with some general notations. Fix a semiring $\boldsymbol{S}$ and a linear function $F : \boldsymbol{S}^N \to \boldsymbol{S}^N$. We can write it as a matrix-vector product: $F(X) = AX + B$, where A is an $N \times N$ matrix, and X, B, are N-dimensional column vectors. After $q + 1$ iterations, the naïve algorithm computes $F^{(q+1)}(0) = B + AB + A^2B + \cdots + A^qB = A^{(q)}B$, where $A^{(q)} \overset{\text{def}}{=} I_N + A + A^2 + \cdots + A^q$. The naïve algorithm converges in $q + 1$ steps iff F is $q + 1$-stable. A matrix A is called q-stable [33] 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, to determine the runtime of the naïve 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 (this was shown in Reference [32] and also follows from our more general Corollary 5.19). We prove here a result in the special case when the semiring is Trop_p^+ (introduced in Example 2.9), which is p-stable.

Lemma 5.20. *Every $N \times N$ matrix A over Trop_p^+ semiring is $((p + 1)N - 1)$-stable. This bound is tight, i.e., there exist $N \times N$-matrices over Trop_p^+ whose stability index is $(p + 1)N - 1$.*

Proof. We consider the $N \times N$-matrix A over Trop_p^+ as the adjacency matrix of a directed graph G with N vertices and up to $p + 1$ parallel edges from some vertex i to j. Then, A_{ij} is a bag of $p + 1$ numbers representing the costs of $p + 1$ edges from i to j; if fewer than $p + 1$ edges exist from i to j, then we complete the bag with ∞, intuitively saying that no further edge from i to j exists. For instance, $A_{ij} = \{1, 2, 3, \infty, \dots, \infty\}$ means that there are 3 edges from i to j of length 1,2,3, while $A_{ij} = \{\infty, \dots, \infty\}$ means that there is no edge from i to j.

Claim. Let $k \geq 1$, let $B = A^{\ell}$ with $\ell \geq k \cdot N$, and let b denote the minimum element in B_{ij} for $1 \leq i, j \leq N$. Moreover, let $C = I + A + A^2 + \cdots + A^{\ell-1}$. Then, C_{ij} contains at least k elements, which are $\leq b$.

The upper bound of $(N+1)p - 1$ follows immediately from the claim. To prove the claim, observe the following:

- $(A^{\ell})_{ij}$ contains the lengths of the $p+1$ lowest-cost paths in G from i to j consisting of at least kN edges.
- C_{ij} contains the lengths of the $p+1$ lowest-cost paths in G from i to j consisting of at most $kN - 1$ edges.

Consider a cost-b path (not necessarily simple) from i to j in G containing at least kN edges. Every segment of length N on this path contains at least a simple cycle. Removing the cycle yields an ij-path with strictly fewer edges with lower or equal cost. By repeating this process, we conclude that there are at least k different ij-paths of cost at most b with $< kN$ edges. This observation proves the claim and thus the upper bound.

To prove the lower bound, consider the following matrix A:

$$\begin{pmatrix} \infty & A_{12} & \infty & \cdots & \cdots & \infty \\ \infty & \infty & A_{23} & \infty & \cdots & \infty \\ \vdots & & & \ddots & \ddots & \vdots \\ \vdots & & & & \ddots & \infty \\ \infty & \cdots & \cdots & \cdots & \infty & A_{(N-1)N} \\ A_{N1} & \infty & \cdots & \cdots & \infty & \infty \end{pmatrix},$$

where we write ∞ for the bag $\{\infty, \ldots, \infty\}$, i.e., the 0-element of Trop_p^+. And we set $A_{12} = A_{23} = \cdots = A_{(N-1)N} = A_{N1} = \{1, \infty, \ldots, \infty\}$, i.e., the number 1 plus p times ∞. In other words, A is the adjacency matrix of the directed cycle of length N. By Claim above, we get $A_{1N}^{(N-1)} = \{N-1, \infty, \ldots, \infty\}$. That is, there exists one path from vertex 1 to vertex N with $N-1$ edges and this path has cost (= length) $N-1$. Moreover, there is an edge of length 1 back to vertex 1. Hence, the $p+1$ shortest paths from vertex 1 to vertex N are obtained by looping $0, 1, 2, \ldots, p$ times through the vertices $1, \ldots, N$ and finally going from 1 to N along the single shortest path. □

The lemma immediately implies:

Corollary 5.21. *Any linear* datalog° *program over* Trop_p^+*, converges in* $(p+1)N - 1$ *steps, where* $N = |D|^{O(1)}$ *is the number of ground IDB tuples. This bound is tight.*

Second, we consider arbitrary p-stable POPS $\boldsymbol{P}$ where $\cdot$ is strict and prove that every linear datalog° program can be computed in polynomial time. Gaussian elimination method, which coincides with the Floyd-Warshall-Kleene algorithm [52, 72], computes the closure A^* of an $N \times N$ matrix in $O(N^3)$ time in a closed semiring. This immediately extends to a p-stable semiring, since every p-stable semiring is closed, by setting $a^* \overset{\text{def}}{=} a^{(p)}$. Our next theorem proves that essentially the same algorithm also applies to p-stable POPS.

Theorem 5.22. *Let* $\boldsymbol{P}$ *be a p-stable POPS, strict* $(x \cdot \bot = \bot)$*, and assume that both operations* $+$ *and* $\cdot$ *can be computed in constant time. Let* $f_1, \ldots, f_N$ *be N linear functions in N variables. Then,* $\mathsf{lfp}(f_1, \ldots, f_N)$ *can be computed in time* $O(pN + N^3)$*.*

Proof. Recall that a polynomial is defined as a sum of monomials; see Equation (37). Thus, a *linear* polynomial is given by an expression $f(x_1, \ldots, x_N) = \sum_{i \in V} a_i x_i + b$, for some set

$V \subseteq [N]$. In other words, we cannot simply write it as $\sum_{i=1,N} a_i x_i + b$, because if we want to make f independent of x_i, then it is not sufficient to set $a_i = 0$, because $0 \cdot \bot = \bot$ and $x + \bot = \bot$; we need to represent explicitly the set of monomials V in f. Equivalently, f is represented by a list of coefficients. Let $g(X) = \sum_{j \in W} c_j x_j + d$ be another linear function. We denote by $f[g/x_k]$ the linear function obtained by substituting x_k with g in f. More precisely, if $k \notin V$, then $f[g/x_k] = f$, and if $k \in V$, then we replace the term $a_k x_k$ in f by $\sum_{j \in W}(a_k c_j)x_j$. The representation of $f[g/x_k]$ can be computed in time $O(N)$ from the representations of f and g. □

ALGORITHM 2: LinearLFP($f_1, \dots, f_N$)

```
if $N = 0$ then
    return ()
if $f_N$ is independent of $x_N$ then
    $c(x_1, \dots, x_{N-1}) \leftarrow f_N(x_1, \dots, x_{N-1})$
if $f_N = a_{NN} \cdot x_N + b(x_1, \dots, x_{N-1})$ then
    $c(x_1, \dots, x_{N-1}) \leftarrow a_{NN}^{(p)} \cdot b(x_1, \dots, x_{N-1}) + \bot$
$(\bar{x}_1, \dots, \bar{x}_{N-1}) \leftarrow$ LinearLFP($f_1[c/x_N], \dots, f_{N-1}[c/x_N]$);
$\bar{x}_N \leftarrow c(\bar{x}_1, \dots, \bar{x}_{N-1})$;
return $(\bar{x}_1, \dots, \bar{x}_N)$
```

Algorithm 2 proceeds recursively on N. Let $H \stackrel{\text{def}}{=} (f_1, \dots, f_N)$. Considering the last linear function $f_N(x_1, \dots, x_N)$, we write H with some abuse as $H(x_1, \dots, x_N) = (f(x,y), g(x,y))$, where $f \stackrel{\text{def}}{=} (f_1, \dots, f_{N-1})$ and $g \stackrel{\text{def}}{=} f_N$, and similarly $x \stackrel{\text{def}}{=} (x_1, \dots, x_{N-1})$, $y \stackrel{\text{def}}{=} x_N$. Then, we apply Lemma 3.3. Since $\boldsymbol{S}$ is p-stable, then function g_x is $p+1$ stable, hence, to compute the fixpoint using Lemma 3.3, we need to represent the function $c(x) \stackrel{\text{def}}{=} g_x^{(p+1)}(\bot)$. There are two cases. The first is when f_N does not depend on x_N: Then $g(x,y) \equiv g(x)$ does not depend on y, and $g_x(y)$ is a constant function, hence, $c(x) = g_x^{(p+1)}(\bot) = g(x)$. The second is when $f_N = a_{NN} \cdot x_N + b(x_1, \dots, x_{N-1})$ for some $a_{NN} \in \boldsymbol{P}$ and linear function b in $N-1$ variables. In our new notation, $g_x(y) = a_{NN} \cdot y + b(x)$. We thus get

$g_x^{(1)}(\bot) = \bot + b(x)$,
$g_x^{(2)}(\bot) = a_{NN} \cdot (\bot + b(x)) + b(x) = \bot + a_{NN} \cdot b(x) + b(x) = a_{NN}^{(1)} \cdot b(x) + \bot$,
$g_x^{(3)}(\bot) = a_{NN} \cdot (a_{NN}^{(1)} \cdot b(x) + \bot) + b(x) = a_{NN}^{(2)} \cdot b(x) + \bot$, and so on.

Hence, $c(x) = g_x^{(p+1)}(\bot) = a_{NN}^{(p)} \cdot b(x) + \bot$. Next, following Lemma 3.3, we compute the fixpoint of $F(x) \stackrel{\text{def}}{=} f(x, c(x))$: This is done by the algorithm inductively, since both f and x have dimension $N-1$.

Finally, we compute the runtime. We need $O(p)$ operations to compute $a_{NN}^{(p)}$, then $O(N^2)$ operations to compute a representation of the linear function c, then representations of the linear functions $f_1[c/x_N], \dots, f_{N-1}[c/x_N]$. Thus, the total runtime of the algorithm is $\sum_{n \le N} O(p + n^2) = O(pN + N^3)$.

6 SEMI-NAÏVE OPTIMIZATION

In this section, we show that the semi-naïve algorithm for standard datalog can be generalized to datalog$^\circ$ for certain restricted POPS. The naïve algorithm 1 repeatedly applies the immediate consequence operator F and computes $J^{(0)}, J^{(1)}, J^{(2)}, \dots$, where $J^{(t+1)} \stackrel{\text{def}}{=} F(J^{(t)})$. This strategy is inefficient, because all facts discovered at iteration t will be re-discovered at iterations $t+1, t+2, \dots$ The *semi-naïve* optimization consists of a modified program that computes $J^{(t+1)}$ by first computing only the "novel" facts $\delta^{(t)} = F(J^{(t)}) - J^{(t)}$, which are then added to $J^{(t)}$ to form $J^{(t+1)}$. Furthermore, the difference $F(J^{(t)}) - J^{(t)}$ can be computed efficiently, *without* fully evaluating $F(J^{(t)})$, by using techniques from incremental view maintenance.

This section generalizes the semi-naïve algorithm from datalog to datalog$^\circ$. The main problem we encounter is that, while the difference operator is well defined in the Boolean semiring, as $x - y \overset{\text{def}}{=} x \wedge \neg y$, no generic difference operator exists in an arbitrary POPS. To define a difference operator, we will restrict the POPS to be a complete, distributive dioid.

6.1 Complete, Distributive Dioids

A *dioid* is a semiring $\boldsymbol{S} = (S, \oplus, \otimes, 0, 1)$ for which $\oplus$ is idempotent (meaning $a \oplus a = a$). Dioids have many applications in a wide range of areas; see Reference [40] for an extensive coverage. We review here a simple property of dioids:

Proposition 6.1. *Let $(S, \oplus, \otimes, 0, 1)$ be a dioid. Then, the following hold:*

(i) $\boldsymbol{S}$ is naturally ordered, and the natural order coincides with the relation $\sqsubseteq$, defined by $a \sqsubseteq b$ if $a \oplus b = b$.
(ii) $\oplus$ is the same as the least upper bound, $\vee$.

Proof. Recall from Section 2 that the natural order is defined as $a \preceq b$ iff $\exists c : a \oplus c = b$. We first show that $\preceq$ is the same as $\sqsubseteq$. One direction, $a \sqsubseteq b$, implies $a \preceq b$ is obvious. For the converse, assume $a \oplus c = b$. Then, $a \oplus b = a \oplus (a \oplus c) = a \oplus c = b$ due to idempotency, and thus $a \sqsubseteq b$. Since $\preceq$ is a preorder, $\sqsubseteq$ is also a preorder. To make it a partial order, we only need to verify anti-symmetry, which is easily verified:: $a \oplus b = b$ and $a \oplus b = a$ imply $a = b$. We just proved (i).

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

Definition 6.2. A POPS $\boldsymbol{S} = (S, \oplus, \otimes, 0, 1, \sqsubseteq)$ is called a *complete, distributive dioid* if $(S, \oplus, \otimes, 0, 1)$ is a dioid, $\sqsubseteq$ is the dioid's natural order, and the ordered set $(S, \sqsubseteq)$ is a *complete, distributive lattice*, which means that every set $A \subseteq S$ has a greatest lower bound $\bigwedge A$, and $x \vee \bigwedge A = \bigwedge \{x \vee a \mid a \in A\}$. In a complete, distributive dioid, the *difference* operator is defined by

$$b \ominus a \overset{\text{def}}{=} \bigwedge \{c \mid a \oplus c \sqsupseteq b\}. \tag{58}$$

To extend the semi-naïve algorithm to datalog$^\circ$, we require the POPS to be a complete, distributive dioid. There are many examples of complete, distributive dioids: $(2^{\mathbf{U}}, \cup, \cap, \emptyset, \mathbf{U}, \subseteq)$ is a complete, distributive dioid, whose difference operator is exactly set-difference $b - a = \bigcap\{c \mid b \subseteq a \cup c\} = b \setminus a$; $\text{Trop}^+ = (\mathbb{R}_+ \cup \{\infty\}, \min, +, \infty, 0, \geq)$ is also a complete, distributive dioid, whose difference operator is defined by Equation (6) in Section 1.1; and $\mathbb{N} \cup \{\infty\}$ is also a complete, distributive dioid. However, Trop^+_p, $\text{Trop}^+_{\leq\eta}$, $\mathbb{R}_\perp$ are not dioids.

The next lemma proves the only two properties of $\ominus$ that we need for the semi-naïve algorithm: We need the first property in Theorem 6.4 to prove the correctness of the algorithm and the second property in Theorem 6.5 to prove the differential rule.

Lemma 6.3. *Let $\boldsymbol{S} = (S, \oplus, \otimes, 0, 1, \sqsubseteq)$ be a complete, distributive dioid, and let $\ominus$ be defined by Equation (58). The following hold:*

$$\text{if } a \sqsubseteq b : \quad a \oplus (b \ominus a) = b \tag{59}$$

$$(a \oplus b) \ominus (a \oplus c) = b \ominus (a \oplus c). \tag{60}$$

Proof. We start by showing a general inequality:

$$\forall a, b \in S: \quad a \oplus (b \ominus a) \sqsupseteq b. \tag{61}$$

This follows from $a \oplus (b \ominus a) = a \oplus \bigwedge\{x \mid a \oplus x \sqsupseteq b\} = a \vee (\bigwedge\{x \mid a \vee x \sqsupseteq b\}) = \bigwedge\{a \vee x \mid a \vee x \sqsupseteq b\} \sqsupseteq b$.

We prove now Equation (59). Denote by $A = \{x \mid a \oplus x \sqsupseteq b\}$, and observe that $b \in A$ because $a \oplus b \sqsupseteq b$. Therefore, $b \ominus a = \bigwedge A \sqsubseteq b$. We also have $a \sqsubseteq b$ by assumption. It follows that $a \oplus (b \ominus a) \sqsubseteq b \oplus b = b$. This, together with (61), implies Equation (59).

We prove Equation (60). For any x for which $a \oplus c \oplus x \sqsupseteq b$ holds, also $a \oplus a \oplus c \oplus x \sqsupseteq a \oplus b$ holds, which implies $a \oplus c \oplus x \sqsupseteq a \oplus b$, because $\oplus$ is idempotent. Conversely, $a \oplus c \oplus x \sqsupseteq a \oplus b$ implies $a \oplus c \oplus x \sqsupseteq b$, because $a \oplus b \sqsupseteq b$. Hence,

$$(a \oplus b) \ominus (a \oplus c) = \bigwedge\{x \mid a \oplus c \oplus x \sqsupseteq a \oplus b\} = \bigwedge\{x \mid a \oplus c \oplus x \sqsupseteq b\} = b \ominus (a \oplus c). \qquad \square$$

6.2 The Semi-naïve Algorithm

We describe now the semi-naïve algorithm for a program over a complete, distributive dioid. Following the notations in Section 4, we write $F(J)$ for the immediate consequence operator of the program, where J is an instance of the IDB predicates. The semi-naïve algorithm is shown in Algorithm 3. It proceeds almost identically to the naïve algorithm 1 but splits the computation $J \leftarrow F(J)$ into two steps: First compute the difference $F(J) \ominus J$, then add it to J. We prove:

ALGORITHM 3: Semi-naïve Evaluation for datalog$^\circ$

```
$J^{(0)} \leftarrow \mathbf{0}$;
for $t \leftarrow 0$ to $\infty$ do
    $\delta^{(t)} \leftarrow F(J^{(t)}) \ominus J^{(t)}$;            // incremental computation, see Section 6.2
    $J^{(t+1)} \leftarrow J^{(t)} \oplus \delta^{(t)}$;
    if $\delta^{(t)} = \mathbf{0}$ then
        Break
return $J^{(t)}$
```

Theorem 6.4. *Consider a* datalog$^\circ$ *program over a complete, distributive dioid. Then, the Semi-naïve Algorithm 3 returns the same answer as the naïve Algorithm 1.*

Proof. We will use identity (59) in Lemma 6.3. Let $J^{(t)}$ be the sequence of IDB instances computed by the semi-naïve Algorithm 3, and let $\bar{J}^{(t)}$ be the sequence computed by the naïve Algorithm 1. Recall that $\bar{J}^{(t)}$ forms an ω-sequence (see Section 3), meaning $\bar{J}^{(0)} \sqsubseteq \bar{J}^{(1)} \sqsubseteq \bar{J}^{(2)} \sqsubseteq \cdots$ We prove, by induction on t, that $J^{(t)} = \bar{J}^{(t)}$. The claim holds trivially for $t = 0$. Assuming $J^{(t)} = \bar{J}^{(t)}$, we prove $J^{(t+1)} = \bar{J}^{(t+1)}$. This follows from:

$$J^{(t+1)} = J^{(t)} \oplus (F(J^{(t)}) \ominus J^{(t)}) = \bar{J}^{(t)} \oplus (F(\bar{J}^{(t)}) \ominus \bar{J}^{(t)}) = F(\bar{J}^{(t)}) = \bar{J}^{(t+1)}.$$

The first equality is the definition of the semi-naïve algorithm, while the second equality uses the induction hypothesis $J^{(t)} = \bar{J}^{(t)}$. The third equality is based on Equation (59), $\bar{J}^{(t)} \oplus (F(\bar{J}^{(t)}) \ominus \bar{J}^{(t)}) = F(\bar{J}^{(t)})$, which holds, because $\bar{J}^{(t)} \sqsubseteq \bar{J}^{(t+1)} = F(\bar{J}^{(t)})$. Finally, the last equality is by the definition of the naïve algorithm. $\square$

6.3 The Differential Evaluation Rule

As described, the semi-naïve algorithm is no more efficient than the naïve algorithm. Its advantages come from the fact that we can compute the difference

$$\delta^{(t)} \leftarrow F(J^{(t)}) \ominus J^{(t)} \tag{62}$$

incrementally without computing the ICO F. Recall that the datalog$^\circ$ program consists of n rules T_i :- $F_i(T_1, \dots, T_n)$, one for each IDB $T_1, \dots, T_n$, where F_i is a sum-sum-product expression (Equation (26) in Section 4), and the ICO is the vector of the sum-sum-product expressions, $F = (F_1, \dots, F_n)$. The difference (62) consists of computing the following differences, for $i = 1, n$:

$$\delta_i^{(t)} \leftarrow F_i(T_1^{(t)}, \dots, T_n^{(t)}) \ominus T_i^{(t)}. \tag{63}$$

For the purpose of incremental evaluation, we will assume w.l.o.g. that each T_j occurs at most once in any sum-product term of F_i; otherwise, we give each occurrence of T_j in (63) a unique name; see Example 6.6 below for an illustration. Therefore, F_i is affine[9] in each argument T_j. Notice that, when $t \geq 1$, then $T_j^{(t)} = T_j^{(t-1)} \oplus \delta_j^{(t-1)}$ for $j = 1, n$, and $T_i^{(t)} = F_i(T_1^{(t-1)}, \dots, T_n^{(t-1)})$. We prove:

Theorem 6.5 (The Differential Rule). *Assume that the sum-sum-product expression $F_i(T_1, \dots, T_n)$ is affine in each argument T_j. Then, when $t \geq 1$, the difference (63) can be computed as follows:*

$$\delta_i^{(t)} \leftarrow \left(\bigoplus_{j=1,n} F_i(T_1^{(t)}, \dots, T_{j-1}^{(t)}, \delta_j^{(t-1)}, T_{j+1}^{(t-1)}, \dots, T_n^{(t-1)}) \right) \ominus T_i^{(t)}. \tag{64}$$

Furthermore, if the sum-sum-product F_i can be written as $F_i(T_1, \dots, T_n) = E_i \oplus G_i(T_1, \dots, T_n)$, where E_i is independent of $T_1, \dots, T_n$ (i.e., it depends only on the EDBs), then Equation (64) can be further simplified to:

$$\delta_i^{(t)} \leftarrow \left(\bigoplus_{j=1,n} G_i(T_1^{(t)}, \dots, T_{j-1}^{(t)}, \delta_j^{(t-1)}, T_{j+1}^{(t-1)}, \dots, T_n^{(t-1)}) \right) \ominus T_i^{(t)}. \tag{65}$$

The significance of the theorem is that it replaces the expensive computation $F_i(T_1^{(t)}, \dots, T_n^{(t)})$ in (63) with several computations $F_i(\cdots)$ (or $G_i(\cdots)$), where one argument is δ_j instead of T_j. This is usually more efficient, because δ_j is much smaller than T_j.

Proof. To reduce clutter, we will write T_j for $T_j^{(t-1)}$ and δ_j for $\delta_J^{(t-1)}$. Therefore, $T_j^{(t)} = T_j \oplus \delta_j$, and the difference (63) becomes:

$$\delta_i^{(t)} \leftarrow F_i(T_1 \oplus \delta_1, \dots, T_n \oplus \delta_n) \ominus F_i(T_1, \dots, T_n). \tag{66}$$

Fix one IDB predicate T_j. Since F_i is affine in T_j, we can write it as $F_j(\dots, T_j, \dots) = E_{ij} \oplus G_{ij}(T_j)$, where E_{ij} does not contain T_j, and each sum-product in G_{ij} contains exactly one occurrence of T_j. This representation of F_i depends on the IDB T_j, hence, we index E_{ij}, G_{ij} by both i and j. Therefore,

$$\begin{aligned} F_i(\dots, T_j \oplus \delta_j, \dots) &= E_{ij} \oplus G_{ij}(T_j \oplus \delta_j) = E_{ij} \oplus G_{ij}(T_j) \oplus G_{ij}(\delta_j) \\ &= (E_{ij} \oplus G_{ij}(T_j)) \oplus (E_{ij} \oplus G_{ij}(\delta_j)) = F_i(\dots, T_j, \dots) \oplus F_i(\dots, \delta_j, \dots), \end{aligned}$$

because $\oplus$ is idempotent, $E_{ij} \oplus E_{ij} = E_{ij}$. Therefore, we have:

$$\begin{aligned} F_i(T_1 \oplus \delta_1, \dots, T_n \oplus \delta_n) = & F_i(T_1, T_2, \dots, T_n) \\ & \oplus F_i(\delta_1, T_2, \dots, T_n) \\ & \qquad \oplus F_i(T_1 \oplus \delta_1, \delta_2, \dots, T_n) \\ & \qquad \dots \\ & \qquad \oplus F_i(T_1 \oplus \delta_1, T_2 \oplus \delta_2, \dots, \delta_n). \end{aligned}$$

[9]In the datalog context, "linear" is often used instead of "affine."

We substitute this in the difference (66), then cancel the first term $F_i(T_1, \ldots, T_n)$ by using identity (60) and obtain

$$\delta_i^{(t)} \leftarrow \left(\bigoplus_{j=1,n} F_i(T_1 \oplus \delta_1, \ldots, T_{j-1} \oplus \delta_{j-1}, \delta_j, T_{j+1}, \ldots, T_n) \right) \ominus F_i(T_1, \ldots, T_n).$$

This proves equality (64).

To prove (65), we observe that both $F_i(T_1^{(t)}, \ldots, T_{j-1}^{(t)}, \delta_j^{(t-1)}, T_{j+1}^{(t-1)}, \ldots, T_j^{(t-1)}) = E_i \oplus G_i(\cdots)$ and $T_i^{(t)} = E_i \oplus G_i(\cdots)$ contain the common term E_i, and therefore, we can use identity (60) to cancel E_i, replacing each F_i with G_i. □

We end this section with a short example.

Example 6.6 (Quadratic Transitive Closure). Consider the following non-linear datalog program computing the transitive closure:

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

The semi-naïve algorithm with the differential rule (65) proceeds as follows: It initializes $T^{(0)} = \emptyset$ and $\delta^{(0)} = E$, then repeats the following instructions for $t = 1, 2, \ldots$:

$$\delta^{(t)}(x, y) \leftarrow \left(\exists z : \delta^{(t-1)}(x, z) \wedge T^{(t-1)}(z, y)\right) \vee \left(\exists z : T^{(t)}(x, z) \wedge \delta^{(t-1)}(z, y)\right) \setminus T^{(t)}(x, y)$$
$$T^{(t+1)}(x, y) \leftarrow T^{(t)}(x, y) \vee \delta^{(t)}(x, y).$$

6.4 Discussion

The semi-naïve algorithm immediately extends to stratified datalog$^\circ$ (see Section 4.5) by applying the algorithm to each stratum.

Our differential rule (64) or (65) is not the only possibility for incremental evaluation. It has the advantage that it requires only n computations of the sum-sum-product expression F_i, where n is the maximum number of IDBs that occur in any sum-product of F_i. But these expressions use both the previous IDBs $T_j^{(t-1)}$ and current IDBs $T_j^{(t)}$, which, one can argue, is not ideal, because the newer IDB instances are slightly larger than the previous ones. An alternative is to use the $2^n - 1$ discrete differentials of F_i, which use only the previous IDBs $T_j^{(t-1)}$, at the cost of increasing the number of expressions. Yet a more sophisticated possibility is to use higher-order differentials, as pioneered by the DBToaster project [45].

The techniques described in this section required the POPS to be a complete, distributive dioid. The attentive reader may have observed that the semi-naïve Algorithm 3 only needs to compute a difference $b \ominus a$ when $a \sqsubseteq b$. This opens the question whether the semi-naïve algorithm can be extended beyond complete, distributive dioids, with a more restricted definition of $\ominus$. We leave this for future work.

7 POPS AND THE WELL-FOUNDED MODEL

Our main motivation in this article was to extend datalog to allow the interleaving of aggregates and recursion. We have seen that by choosing different POPS, one can capture different kinds of aggregations. A large literature on datalog is concerned, however, with a different extension: the interleaving of negation and recursion. It turns out that a certain POPS can also be used to interpret datalog programs with negation; we give the details in this section.

Let us briefly review the most important approaches to extend datalog with negation. The simplest is *stratified datalog*, which is the most commonly used in practice, but has limited expressive

$$\begin{tikzcd}
a \arrow[r] \arrow[d, shift right] & c \arrow[rr] \arrow[dr] & & e \arrow[r] & f \\
b \arrow[u, shift right] & & d \arrow[ur] & &
\end{tikzcd}$$

Fig. 4. Example graph with edges $E = \{(a, b), (a, c), (b, a), (c, d), (c, e), (d, e), (e, f)\}$, used for the win-move game.

power [48]. Gelfond and Lifschitz [30] introduced *stable model semantics*, which has been used in constraint satisfaction systems like DLV [53] but has high data complexity, making it unsuitable for large-scale data analytics. One possibility is to simply allow the ICO to be non-monotone [49], but this leads to an operational, non-declarative semantics. The extension most related to our article is the *well-founded semantics*, introduced by Van Gelder et al. [29] and refined in References [21, 23, 28, 67] (see the survey cited in Reference [24]). This semantics can be described in terms of an alternating fixpoint [28], and therefore it is tractable. Przymusinski [68] extended the semantics from 2-valued to 3-valued (thus, allowing atoms to be true, false, or undefined) and proved that every program has a 3-valued well-founded model that is also the minimal 3-valued stable model. Fitting [20] extended this result to allow 4-valued logics (to handle contradiction) and other bilattices. In this section, we briefly review the well-founded model and illustrate its connection to the least fixpoint semantics of datalog$^\circ$ on the POPS THREE, introduced in Section 2.5.2.

7.1 Review of the Well-founded Model

Instead of a formal treatment, we prefer to describe the well-founded model on a particular example: the win-move game, which has been extensively studied in the literature as a prototypical datalog program with negation that cannot be stratified [2]. Two players play the following pebble game on a graph $E(X, Y)$. The first player places the pebble on some node. Players take turns and move the pebble, which can only be moved along an edge. A player who cannot move loses. The problem is to compute the set of winning position for the first player. This set is expressed concisely by the following rule:

$$Win(X) \text{ :- } \exists_Y (E(X, Y) \wedge \neg Win(Y)). \tag{67}$$

This is a datalog program *with* negation. Its ICO is no longer monotone, and the standard least fixpoint semantics no longer applies.

The well-founded model circumvents this problem. It admits several, equivalent definitions; we follow here the alternating fixpoint semantics from Reference [28]. Consider the grounding of the win-move program based on the graph shown in Figure 4. To reduce clutter, we write $W(X)$ for $Win(X)$, and $\overline{W}(X)$ for $\texttt{not}(Win(X))$. We also write "=" for :- . Since $E(X, Y) = 1$ whenever the edge (X, Y) is present and $E(X, Y) = 0$ otherwise, the grounded program is:

$$\begin{aligned}
W(a) &= (E(a,a) \wedge \overline{W}(a)) \vee (E(a,b) \wedge \overline{W}(b)) \vee (E(a,c) \wedge \overline{W}(c)) \vee (E(a,d) \wedge \overline{W}(d)) \vee (E(a,e) \wedge \overline{W}(e)) \vee (E(a,f) \wedge \overline{W}(f)) \\
&= \overline{W}(b) \vee \overline{W}(c) \\
W(b) &= (E(b,a) \wedge \overline{W}(a)) \vee (E(b,b) \wedge \overline{W}(b)) \vee (E(b,c) \wedge \overline{W}(c)) \vee (E(b,d) \wedge \overline{W}(d)) \vee (E(b,e) \wedge \overline{W}(e)) \vee (E(b,f) \wedge \overline{W}(f)) \\
&= \overline{W}(a) \\
W(c) &= \overline{W}(d) \vee \overline{W}(e) \\
W(d) &= \overline{W}(e) \\
W(e) &= \overline{W}(f) \\
W(f) &= \mathbf{0}.
\end{aligned}$$

The rule for $W(a)$ was obtained by substituting the existential variable Y with each node a, b, c, d, e, then we simplified it, noting that $E(a, b) = E(a, c) = 1$ and $E(a, a) = E(a, d) = E(a, e) = 0$; similarly for $W(b)$. For the other grounded rules, we show only the final, simplified form.

The alternating fixpoint semantics consists of computing a sequence $J^{(t)}$, $t \geq 0$ of standard, 2-valued IDB instances, where $J^{(0)} = \emptyset$, and $J^{(t+1)}$ is obtained by first replacing each negative atom in every ground rule by its Boolean value according to $J^{(t)}$, then $J^{(t+1)}$ is defined as the least fixpoint of the resulting, monotone program. In the above example, we thus get the following sequence of truth assignments:

		$W(a)$	$W(b)$	$W(c)$	$W(d)$	$W(e)$	$W(f)$		
$J^{(0)}$	$=$	$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$		
$J^{(1)}$	$=$	$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$	$\mathbf{0}$		
$J^{(2)}$	$=$	$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$	$\mathbf{1}$	$\mathbf{0}$		
$J^{(3)}$	$=$	$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$	$\mathbf{0}$	$\mathbf{1}$	$\mathbf{0}$		
$J^{(4)}$	$=$	$\mathbf{0}$	$\mathbf{0}$	$\mathbf{1}$	$\mathbf{0}$	$\mathbf{1}$	$\mathbf{0}$		
$J^{(5)}$	$=$	$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$	$\mathbf{0}$	$\mathbf{1}$	$\mathbf{0}$	$=$	$J^{(3)}$
$J^{(6)}$	$=$	$\mathbf{0}$	$\mathbf{0}$	$\mathbf{1}$	$\mathbf{0}$	$\mathbf{1}$	$\mathbf{0}$	$=$	$J^{(4)}$.

The following holds: $J^{(0)} \subseteq J^{(2)} \subseteq J^{(4)} \subseteq \cdots \subseteq J^{(5)} \subseteq J^{(3)} \subseteq J^{(1)}$. In other words, the even-numbered instances form an increasing chain, while the odd-numbered instances form a decreasing chain. Their limits, denote them by $L = \bigcup_t J^{(2t)}$ and $G = \bigcap_t J^{(2t+1)}$, are the least fixpoint and the greatest fixpoint, respectively, of a certain monotone program; see Reference [28]. The well-founded model is defined as consisting of all positive literals in L and all negative literals missing from G. In our example, the well-founded model is $\{W(c), W(e), \overline{W}(d), \overline{W}(f)\}$, because $L = J^{(4)} = \{W(c), W(e)\}$ and $G = J^{(3)} = \{W(a), W(b), W(c), W(e)\}$, and its complement is $\{W(d), W(f)\}$. Alternatively, the well-founded model can be described as assigning the value 1 to the atoms $W(c), W(e)$, the value 0 to the atoms $W(d), W(f)$, and the value $\bot$ (undefined) to $W(a), W(b)$.

7.2 The POPS THREE

Building on earlier work by Przymusinski [68], Fitting [20] describes the following three-valued semantics of logic programs with negation: Starting from Kleene's 3-valued logic $\{0, 0.5, 0\}$, where $\vee, \wedge, \neg$ are $\max, \min, 1 - x$, respectively [22], he defined two order relations on the set THREE $= \{\bot, 0, 1\}$:

The *truth order:*	$0 \leq_t \bot \leq_t 1$	
The *knowledge order:*	$\bot \leq_k 0$	$\bot \leq_k 1$.

The truth order is the same order as in Kleene's 3-valued logic, and, for this purpose, the value $\bot$ can be interpreted as 0.5. This is also the standard 3-valued logic used in SQL. Fitting argued that, instead of interpreting logic programs using the truth order, where negation is not monotone, one should interpret them using the knowledge order, where negation is monotone.

Fitting's semantics coincides, sometimes, with the well-founded model. It also coincides with the least fixpoint of datalog$^\circ$ interpreted over the POPS THREE, which we illustrate next. Recall from Section 2.5.2 that $\texttt{THREE} \stackrel{\text{def}}{=} (\{\bot, 0, 1\}, \vee, \wedge, 0, 1, \leq_k)$ is a POPS where the semiring operations are the lub $\vee$ and the glb $\wedge$ of the truth order, and where the order relation is the knowledge order $\leq_k$. In particular, we have $x \wedge \mathbf{0} = \mathbf{0}$ and $x \vee \mathbf{1} = \mathbf{1}$ for every x including $x = \bot$. Hence, $\wedge$ is absorbing, i.e., THREE constitutes a semiring and should not be confused with the POPS $\mathbb{B}_\bot \stackrel{\text{def}}{=} \mathbb{B} \cup \{\bot\}$ that we get by lifting the Booleans analogously to lifting the integers and reals to $\mathbb{N}_\bot$ and $\mathbb{R}_\bot$, respectively.

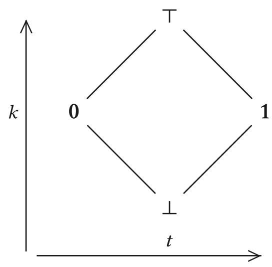


Fig. 5. Orders $\leq_k$ and $\leq_t$ in the bilattice FOUR [21].

The following datalog° program is the win-move game (67) over the POPS THREE:

$$Win(Y) \text{ :- } \bigoplus_Y E(X,Y) \wedge \texttt{not}(Win(Y)).$$

Here, not is a monotone function w.r.t. $\leq_k$ defined as $\texttt{not}(\mathbf{0}) = \mathbf{1}$, $\texttt{not}(\mathbf{1}) = \mathbf{0}$, and $\texttt{not}(\bot) = \bot$. The fixpoint semantics computes a sequence of IDB instances $W^{(0)} \sqsubseteq W^{(1)} \sqsubseteq W^{(2)} \sqsubseteq \cdots$, shown here:

		$W(a)$	$W(b)$	$W(c)$	$W(d)$	$W(e)$	$W(f)$	
$W^{(0)}$	=	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	
$W^{(1)}$	=	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\mathbf{0}$	
$W^{(2)}$	=	$\bot$	$\bot$	$\bot$	$\bot$	$\mathbf{1}$	$\mathbf{0}$	
$W^{(3)}$	=	$\bot$	$\bot$	$\bot$	$\mathbf{0}$	$\mathbf{1}$	$\mathbf{0}$	
$W^{(4)}$	=	$\bot$	$\bot$	$\mathbf{1}$	$\mathbf{0}$	$\mathbf{1}$	$\mathbf{0}$	$= W^{(5)}$.

The least fixpoint $W^{(4)}$ is the same as the well-founded model of the win-move program (67).

7.3 Discussion

It should be noted that Fitting's 3-valued interpretation of logic programs (and, likewise, datalog° over the POPS THREE) does not always yield the well-founded semantics. In fact, in general, it does not even coincide with the minimal model semantics of datalog without negation. To see this, consider the following datalog program (or, equivalently, datalog° program):

$$P(a) \text{ :- } P(a).$$

In the minimal model of this datalog program (and, equivalently, when considering this datalog° program over the POPS $\mathbb{B}$), we have $P(a) = \mathbf{0}$. In contrast, Fitting's 3-valued semantics (and, equivalently, datalog° over the POPS THREE) yields $P(a) = \bot$. In Reference [24], Fitting asks "Which is the 'right' choice?" and he observes that a case can be made both for setting $P(a) = \mathbf{0}$ and for setting $P(a) = \bot$. He thus concludes that the "intended applications probably should decide."

In References [21, 23], 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, where the additional truth value $\top$ denotes "both false and true." We thus 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 $\bot \leq_k \mathbf{0}, \mathbf{1} \leq_k \top$, which are visualized in Figure 5 (cf. Reference [21], Figure 1). The not function is readily extended to $\top$ by setting $\texttt{not}(\top) = \top$. It can be shown that the additional truth value $\top$ has no effect on the fixpoint iteration w.r.t. $\leq_k$ that we considered above for the POPS THREE. In fact, Fitting showed that $\top$ can never occur as truth value in the least fixpoint w.r.t. $\leq_k$ (cf. Reference[21], Proposition 7.1). We note that FOUR is also a POPS, where the semiring operations $\oplus, \otimes$ are the glb and lub of the truth order, and the partial order of the POPS is the knowledge order.

8 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 (Minimum Spanning Tree)**. Most notably, the non-deterministic *choice* construct was extensively studied early on [34–36]. While datalog+choice is powerful, its expression and semantics are somewhat clunky, geared towards answering optimization questions (greedily). In particular, it was not designed to deal with general aggregations.

To evaluate recursive datalog$^\circ$ program is to solve fixpoint equations over semirings, which was studied in the automata theory [51], program analysis [12, 63], and graph algorithms [9, 55, 56] communities since the 1970s (see References [33, 41, 52, 72, 88] and references therein). The problem took slightly different forms in these domains, but at its core, it is to find a solution to the equation $\mathbf{x} = \mathbf{f}(\mathbf{x})$, where $\mathbf{x} \in S^n$ is a vector over the domain S of a semiring, and $\mathbf{f} : S^n \longrightarrow S^n$ has multivariate polynomial component functions.

When $\mathbf{f}$ is affine (what we called *linear* in this article), researchers realized that many problems in different domains are instances of the same problem, with the same underlying algebraic structure: transitive closure [82], shortest paths [25], Kleene's theorem on finite automata and regular languages [44], continuous dataflow [12, 43], and so on. Furthermore, these problems share the same characteristic as the problem of computing matrix inverse [7, 31, 77]. The problem is called the *algebraic path problem* [72], among other names, and the main task is to solve the matrix fixpoint equation $X = AX + I$ 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 References [33, 72] for more detailed discussions. Here, we briefly summarize the main approaches.

The first approach is to keep iterating 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 $x = 1 + ax$, can solve the algebraic path problem in $O(n^3)$-time [52, 72].

The third approach is based on specifying the solutions based on the free semiring generated when viewing $\mathbf{A}$ as the adjacency matrix of a graph [78]. The underlying graph structure (such as planarity) may sometimes be exploited for very efficient algorithms [55, 56].

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 the Kleene algebra [50, 65, 66]. 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 [13] 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 [19, 41].

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. [19] and Hopkins and Kozen [41] 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's method does. Furthermore, if the semiring is commutative and idempotent (in addition to being ω-continuous), then Newton's 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!

Recently, semi-naïve evaluation has been extended for a higher-order functional language called Datafun [6]. The book cited in Reference [33] contains many fundamental results on algebraic structures related to semirings and computational problems on such structures.

9 CONCLUSIONS

A massive number of application domains demand that we move beyond the confines of the Boolean world: from program analysis [12, 63], graph algorithms [9, 55, 56], provenance [38], formal language theory [51], to machine learning and linear algebra [1, 70]. 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 interpreted function such as sigmoid will allow it to express typical neural network computations. Adding another semiring to the query language (in the sense of FAQ [4]) 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 article established part of the bridge. There are many interesting 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 the size of the input database is open. This is open even for linear programs. Our result on Trop_p in Section 5 indicates that the linear case is likely in PTIME.

We have discussed in Section 4 several extensions to datalog$^\circ$ that we believe are necessary in practice. It remains open whether our convergence results continue to hold for those extensions.

Two of the most important extensions of datalog/logic programming are concerned with negation and aggregation. Various semantics have been proposed in the literature for these extensions; see References [57] and [58] for overviews of the various approaches. As discussed in Section 7, negation can be added to datalog$^\circ$ as an interpreted predicate. In particular, Fitting's three-valued semantics [20] of datalog with negation can thus be naturally captured in datalog$^\circ$ by an appropriate choice of the POPS. However, the question remains if we can also extend results for other semantics (above all well-founded semantics [29] and stable model semantics [30]) from general datalog/logic programming to datalog$^\circ$ with negation.

The extension of datalog with aggregation was one of the original motivations for considering datalog over semirings [71]. Analogously to datalog with negation, also for datalog with aggregation, several semantics have been proposed [58]. Again, it remains a question for future work if we can extend our results on datalog$^\circ$ to the various semantics of datalog with aggregation.

More generally, a systematic study of the expressive power of datalog$^\circ$ is left for future work. While we have powerful tools (see, e.g., Reference [54]) for analyzing the expressive power of various logics (or, equivalently, of various query languages over the Boolean semiring) and pinpointing their limits, the formal study of the expressive of query languages over semirings has barely started. In their recent work [8], Brinke et al. have extended Ehrenfeucht-Fraïssé games to first-order logic over semirings. The extension of such tools to other query languages over semirings, including datalog$^\circ$, would be a natural next step.

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 trade off 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.

APPENDICES

A LOWER BOUND FOR THEOREM 3.4

Proof of the Lower Bound. To keep the notation simple, we will choose $p_1 = p_2 = \cdots = p_n = p$ below. It is straightforward to extend this case to arbitrary values $p_1, p_2, \ldots, p_n$. Moreover, we may restrict ourselves to the case $p \geq 1$, since the lower bound for $p = 0$ is trivial. We define n posets $\mathbf{L}_1, \ldots, \mathbf{L}_n$, and functions $f_1, \ldots, f_n$ of types $f_i : \prod_{j=1,n} \mathbf{L}_j \longrightarrow \mathbf{L}_i$, as follows: For every $i \in [n]$, let $\mathbf{L}_i \stackrel{\text{def}}{=} \{a_j^{(i)} \mid j \in \mathbb{N}\}$ with the linear order $a_0^{(i)} < a_1^{(i)} < a_2^{(i)} < \cdots$. By slight abuse, we will simply write $\mathbf{L}_i = \mathbb{N}$ with the understanding that, in $\mathbf{L}_i$, integer j actually stands for $a_j^{(i)}$. Only when it comes to the definition of the c-clone and we have to take care of type-compatibilities, it will be important to keep in mind that the $\mathbf{L}_i$'s are pairwise distinct.

For every $i \in [n]$, we define the function $f_i : \prod_{j=1,n} \mathbf{L}_j \longrightarrow \mathbf{L}_i$ by setting $f_i(c_1, \ldots, c_n) = d_i$ via the following case distinction:

(1) Case $i < n$:
   (a) if $c_{i+1} \geq p^{i+1}$ then $d_i = p^i$;
   (b) else $d_i = \lfloor c_{i+1}/p \rfloor$.
(2) Case $i = n$:
   (a) if there exists $j \in [n]$ with $c_j \geq p^j$ then $d_n = p^n$;
   (b) else if $c_n \geq (c_{n-1} + 1) \cdot p$ then $d_n = c_n$;
   (c) else if $c_n > c_{n-1} \cdot p$ then $d_n = c_n + 1$;
   (d) else if there exists $j < n - 1$ with $c_j \cdot p^{n-j} < c_{n-1} \cdot p$ then $d_n = c_{n-1} \cdot p$;
   (e) else $d_n = c_{n-1} \cdot p + 1$;

Let $h = (f_1, \ldots, f_n)$ and $m \geq 0$. It is easy to verify that the sequence $h^{(m)}(0, \ldots, 0)$ with $h = (f_1, \ldots, f_n)$ and $m \geq 0$ consists of all n-tuples $(c_1, \ldots, c_n)$ with $0 \leq c_i \leq p^i$ in ascending lexicographical order. The sequence is depicted in Figure 6. For every i, the ith component is incremented until the limit p^i is reached. Moreover, with each application of h, we only increment one component. Hence, function h indeed has stability index $p + p^2 + p^3 + \cdots + p^n$.

It remains to show that the functions $h, f_1, \ldots, f_n$ are part of a c-clone C, such that every function $f : \mathbf{L}_i \longrightarrow \mathbf{L}_i$ in C is p-stable. We choose C as the smallest c-clone containing the functions $h, f_1, \ldots, f_n$. We have to show that every function $f : \mathbf{L}_i \longrightarrow \mathbf{L}_i$ in C is p-stable and that all functions in C are monotone.

Intuition. Before we prove these properties, we discuss the rationale behind the above definition of the functions f_i. The main challenge here is, that the functions f_i have to be defined over the entire domain, i.e., we also have to consider value combinations $(c_1, \ldots, c_n)$ that are never reached by $h^{(m)}(0, \ldots, 0)$. An inspection of the sequence $h^{(m)}(0, \ldots, 0)$ shows that, for every $i < n$, component c_{i+1} may only range from $c_i \cdot p$ to $(c_i + 1) \cdot p$. Equivalently, c_i is fully determined by the value of c_{i+1}, namely, $c_i = \lfloor c_{i+1}/p \rfloor$. Hence, in Case 1.b, we set $f_i(c_1, \ldots, c_n) = \lfloor c_{i+1}/p \rfloor$. However, in Case 1.a, we first have to make sure that $f_i(c_1, \ldots, c_n)$ never takes a value above p^i.

$$\begin{array}{ccc}
(0,\ldots,0,0), & \cdots & (0,\ldots,0,p),\\
(0,\ldots,1,p), & \cdots & (0,\ldots,1,2p),\\
\vdots & & \vdots\\
(0,\ldots,p-1,p^2-p), & \cdots & (0,\ldots,p-1,p^2),\\
(0,\ldots,0,p,p^2), & & \\
(0,\ldots,1,p,p^2), & \cdots & (0,\ldots,1,p,p^2+p),\\
(0,\ldots,1,p+1,p^2+p), & \cdots & (0,\ldots,1,p,p^2+2p),\\
\vdots & & \vdots\\
(0,\ldots,p-1,p^2-1,p^3-p), & \cdots & (0,\ldots,p-1,p^2-1,p^3),\\
(0,\ldots,0,p-1,p^2,p^3), & & \\
(0,\ldots,0,p,p^2,p^3), & & \\
(0,\ldots,1,p,p^2,p^3), & \cdots & (0,\ldots,1,p,p^2,p^3+p),\\
\vdots & & \vdots\\
(p-1,p^2-1,\ldots, & & (p-1,p^2-1,\ldots,\\
\quad p^{n-1}-1,p^n-p), & \cdots & \quad p^{n-1}-1,p^n),\\
(p-1,p^2-1,\ldots,p^{n-1},p^n), & & \\
\vdots & & \\
(p-1,p^2,\ldots,p^{n-1},p^n), & & \\
(p,p^2,\ldots,p^{n-1},p^n). & &
\end{array}$$

Fig. 6. Sequence of tuples produced by $h^{(m)}(0,\ldots,0)$.

Similarly, in Case 2.a, whenever one of c_j has reached its upper bound p^j (or is even greater), then we also set $f_n(c_1,\ldots,c_n)$ to its upper bound, i.e., p^n. Of course, in the sequence produced by $h^{(m)}(0,\ldots,0)$, the jth component can never take a value $> p^j$ and also the value p^j itself is only possible if the nth component has already reached p^n.

After having excluded the case of an excessively big component c_j in Case 2.a, we still have to distinguish 4 further cases: We have already observed above that the $(i+1)$-st component can only take values in the interval $[c_i \cdot p, (c_i+1)\cdot p]$. That is, c_n may only take values in $[c_{n-1}\cdot p, (c_{n-1}+1)\cdot p]$. Hence, if $c_n \geq (c_{n-1}+1)\cdot p$, then we must not further increase c_n. This is taken care of by Case 2.b. In fact, in the sequence $h^{(m)}(0,\ldots,0)$, we first have to increase c_{n-1} and possibly further components to the left, before we may continue incrementing c_n. The incrementation of c_n, as long as c_n is in the allowed range, is realized via Case 2.c.

Cases 2.d and 2e only apply if $c_n \leq c_{n-1}\cdot p$ holds. Both cases cover two kinds of situations: $c_n = c_{n-1}\cdot p$, which may occur in the sequence $h^{(m)}(0,\ldots,0)$, and $c_n < c_{n-1}\cdot p$, which cannot occur in this sequence. For the case $c_n = c_{n-1}\cdot p$, recall that c_n may only take values in $[c_{n-1}\cdot p, (c_{n-1}+1)\cdot p]$. An inspection of the sequence $h^{(m)}(0,\ldots,0)$ shows that also the components c_j with $j < n-1$ impose a restriction on the allowed values of c_n, namely: for all $j < n-1$, component c_n must be in the interval $[c_j \cdot p^{n-j}, (c_j+1)\cdot p^{n-j}]$. Hence, if $c_n = c_{n-1}\cdot p$, then we have to distinguish the two cases, if we must keep c_n unchanged and first increment a component c_j to the left of c_{n-1} (i.e., Case 2.d) or if we may continue incrementing c_n (i.e., Case 2.e).

Cases 2.d and 2e also cover the case $c_n < c_{n-1}\cdot p$, which can never occur in $h^{(m)}(0,\ldots,0)$. When defining $f_n(c_1,\ldots,c_n)$ in this case, we have to make sure that f_n is monotone and every function $f : \mathbf{L}_n \longrightarrow \mathbf{L}_n$ in C is p-stable. This goal is achieved via Cases 2.d and 2e by treating $c_n < c_{n-1}\cdot p$ in the same way as $c_n = c_{n-1}\cdot p$. The following example illustrates why this makes sense: let $p = 3$ and $n = 3$, and consider the tuple $(2, p, 0)$. The third component $c_3 = 0$ is clearly below the interval of allowed values $[p^2, p^2+p]$ by $c_2 = p$. When choosing an appropriate value for $f_3(2, p, 0)$, we must be careful so as not to destroy monotonicity. In particular, by $f_3(0,0,0) = 1$, we have to choose $f_3(2,p,0) \geq 1$. However, when choosing a concrete value ≥ 1, we also have to keep the p-stability of all functions $f : \mathbf{L}_n \to \mathbf{L}_n$ in C in mind. In particular, consider the function f obtained from f_3 when fixing the first two components to constants 2 and p, i.e., $f(x) = f_3(2,p,x)$. We cannot simply set $f_3(2,p,0) = 1$, i.e., $f(0) = 1$ for the following reason: consider the tuple $(2,p,1)$. By monotonicity, we have to set $f(1) = f_3((2,p,1) \geq f_3(0,0,1) = 2$. This argument can be repeated for $f_3(2,p,x)$ for every $x \leq p^2$. To ensure p-stability of f, we have to choose $f_3(2,p,0)$ big enough so at most $p-1$ further incrementations of c_3 are possible. That is why we simply treat $f_3(2,p,0)$ in the same way as $f_3(2,p,p^2)$.

We now prove that C defined as the smallest c-clone containing functions $h, f_1, \ldots, f_n$ has the desired properties in terms of monotonicity and p-stability:

Proof of Monotonicity. We show that every function f_i is monotone. From this, it follows easily that all functions in C are monotone. The case $i < n$ will turn out rather simple. In contrast, the case $i = n$ will require a more detailed analysis. The following observation will be helpful: no matter which of the Cases 2.b–2.e is used to define $f_n(c_1, \ldots, c_n)$, the inequations $f_n(c_1, \ldots, c_n) \leq p^n$ and $f_n(c_1, \ldots, c_n) \geq c_n$ and $f_n(c_1, \ldots, c_n) \geq c_{n-1} \cdot p$ always hold. This property is easy to verify by inspecting the definition of $f_n(c_1, \ldots, c_n)$ in each of the four Cases 2.b–2.e.

Now consider two tuples $(c_1, \ldots, c_n)$ and $(c'_1, \ldots, c'_n)$ with the property $(c_1, \ldots, c_n) \sqsubseteq (c'_1, \ldots, c'_n)$, i.e., $c_j \leq c'_j$ for every j. To show $f_i(c_1, \ldots, c_n) \leq f_i(c'_1, \ldots, c'_n)$, we distinguish, which case of the definition of f_i applies to $(c_1, \ldots, c_n)$.

Case 1: $i < n$. If (1.a) $c_{i+1} \geq p^{i+1}$ then also $c'_{i+1} \geq p^{i+1}$ and we have $f_i(c_1, \ldots, c_n) = p^i = f_i(c'_1, \ldots, c'_n)$. However, if (1.b) $c_{i+1} < p^{i+1}$, then $f_i(c_1, \ldots, c_n) = \lfloor c_{i+1}/p \rfloor < p^i$. Hence, no matter whether $f_i(c'_1, \ldots, c'_n)$ is determined via Case 1.a or 1.b, the property $f_i(c_1, \ldots, c_n) \leq f_i(c'_1, \ldots, c'_n)$ is always guaranteed.

Case 2: $i = n$. We analyze each of the Cases 2.a–2.e separately.

Case 2.a. If there exists $j \in [n]$ with $c_j \geq p^j$ then also $c'_j \geq p^j$ and we have $f_n(c_1, \ldots, c_n) = p^n = f_n(c'_1, \ldots, c'_n)$.

Case 2.b. If $c_n \geq (c_{n-1}+1) \cdot p$ then $f_n(c_1, \ldots, c_n) = c_n \leq p^n$. However, we either have $f_n(c'_1, \ldots, c'_n) = p^n$ (if Case 2.a applies) or $f_n(c'_1, \ldots, c'_n) \geq c'_n$ (in all other cases) by the above observation. Hence, $f_n(c_1, \ldots, c_n) \leq f_n(c'_1, \ldots, c'_n)$ clearly holds.

Case 2.c. If $c_n > c_{n-1} \cdot p$ then $f_n(c_1, \ldots, c_n) = c_n + 1 \leq p^n$. Again, recall that we either have $f_n(c'_1, \ldots, c'_n) = p^n$ (if Case 2.a applies) or $f_n(c'_1, \ldots, c'_n) \geq c'_n$ (in all other cases). Hence, the only interesting case is that Case 2.a does not apply to $f_n(c'_1, \ldots, c'_n)$ and $c'_n = c_n$. We now show that, no matter which of the remaining Cases 2.b–2.e applies to $f_n(c'_1, \ldots, c'_n)$, we always get the desired inequation $f_n(c_1, \ldots, c_n) \leq f_n(c'_1, \ldots, c'_n)$.

Note that we are dealing with the case $c_n < (c_{n-1}+1) \cdot p$. Moreover, we are assuming $c'_n = c_n$ and $c'_{n-1} \geq c_{n-1}$. Hence, we also have $c'_n < (c'_{n-1}+1) \cdot p$, which means that $f_n(c'_1, \ldots, c'_n)$ cannot be defined via Case 2.b.

If $f_n(c'_1, \ldots, c'_n)$ is defined via Case 2.c, then $f_n(c'_1, \ldots, c'_n) = c'_n + 1$ holds and we are done.

If one of the Cases 2.d or 2.e applies, then $c'_{n-1} \cdot p \geq c'_n$ must hold. In contrast, we are considering the Case 2.c for $f_n(c_1, \ldots, c_n)$. That is, $c_{n-1} \cdot p < c_n$. Since we are assuming $c'_n = c_n$, we have $c'_{n-1} > c_{n-1}$ or, equivalently, $c'_{n-1} \geq c_{n-1} + 1$. By the above observation, we have $f_n(c'_1, \ldots, c'_n) \geq c'_{n-1} \cdot p$. However, since Case 2.b does not apply to $f_n(c_1, \ldots, c_n)$, we have $c_n < (c_{n-1}+1) \cdot p$ or, equivalently, $c_n + 1 \leq (c_{n-1}+1) \cdot p$. In total, this gives $f_n(c'_1, \ldots, c'_n) \geq c'_{n-1} \cdot p \geq (c_{n-1}+1) \cdot p \geq c_n + 1 = f_n(c_1, \ldots, c_n)$.

Case 2.d. If there exists $j < n-1$ with $c_j \cdot p^{n-j} < c_{n-1} \cdot p$ then $f_n(c_1, \ldots, c_n) = c_{n-1} \cdot p \leq p^n$. However, we either have $f_n(c'_1, \ldots, c'_n) = p^n$ (if Case 2.a applies) or $f_n(c'_1, \ldots, c'_n) \geq c'_{n-1} \cdot$

p (in all other cases) by the above observation. Either way, $f_n(c_1, \dots, c_n) \leq f_n(c'_1, \dots, c'_n)$ holds.

Case 2.e. If for all $j < n-1$: $c_j \cdot p^{n-j} \geq c_{n-1} \cdot p$ then $f_n(c_1, \dots, c_n) = c_{n-1} \cdot p + 1 \leq p^n$. By the above observation, either $f_n(c'_1, \dots, c'_n) = p^n$ holds (if Case 2.a applies) or $f_n(c'_1, \dots, c'_n) \geq c'_{n-1} \cdot p$ (in all other cases) holds. Hence, the only interesting case is that Case 2.a does not apply to $f_n(c'_1, \dots, c'_n)$ and $c'_{n-1} = c_{n-1}$. We now show that, no matter which of the remaining Cases 2.b–2.e applies to $f_n(c'_1, \dots, c'_n)$, we always get the desired inequation $f_n(c_1, \dots, c_n) \leq f_n(c'_1, \dots, c'_n)$.

Case 2.b applies only if $c'_n \geq (c'_{n-1} + 1) \cdot p$, thus $f_n(c'_1, \dots, c'_n) = c'_n$. We are assuming $p \geq 1$; hence, $(c'_{n-1} + 1) \cdot p \geq c'_{n-1} \cdot p + 1$ holds. In total, we thus have $f_n(c'_1, \dots, c'_n) = c'_n \geq c'_{n-1} \cdot p + 1 \geq c_{n-1} \cdot p + 1 = f_n(c_1, \dots, c_n)$.

If Case 2.c applies, then $c'_n > c'_{n-1} \cdot p$ and we set $f_n(c'_1, \dots, c'_n) = c'_n + 1$. Hence, we have $f_n(c'_1, \dots, c'_n) = c'_n + 1 > c'_{n-1} \cdot p + 1 \geq c_{n-1} \cdot p + 1 = f_n(c_1, \dots, c_n)$.

Note that Case 2.d would mean that there exists $j < n-1$ with $c'_j \cdot p^{n-j} < c'_{n-1} \cdot p$. However, this cannot happen by our current assumptions, namely: for every j: $c'_j \geq c_j$, $c_{n-1} = c'_{n-1}$, and for all $j < n-1$: $c_j \cdot p^{n-j} \geq c_{n-1} \cdot p$.

Finally, if Case 2.e applies, then $f_n(c'_1, \dots, c'_n) = c'_{n-1} \cdot p + 1$. Together with $c'_{n-1} = c_{n-1}$ and $f_n(c_1, \dots, c_n) = c_{n-1} \cdot p + 1$, we again have $f_n(c_1, \dots, c_n) \leq f_n(c'_1, \dots, c'_n)$.

Proof of Stability. By choosing C as the minimal c-clone containing the functions $h, f_1, \dots, f_n$, we know that all functions $f : \mathbf{L}_i \longrightarrow \mathbf{L}_i$ are of the form $f_{\mathbf{c}} : \mathbf{L}_i \longrightarrow \mathbf{L}_i$ with $\mathbf{c} = (c_1, \dots, c_{i-1}, c_{i+1}, \dots, c_n)$ and $f_{\mathbf{c}}(x) = f_i(c_1, \dots, c_{i-1}, x, c_{i+1}, \dots, c_n)$ for some constants $c_j \in \mathbf{L}_j$ for every $j \neq i$. We have to show that $f_{\mathbf{c}}^{(p)}(0) = f_{\mathbf{c}}^{(p+1)}(0)$ holds. To this end, we distinguish which of the cases of the definition of f_i applies to $(c_1, \dots, c_{i-1}, 0, c_{i+1}, \dots, c_n)$.

Case 1: $i < n$. If (1.a) $c_{i+1} \geq p^{i+1}$ then $f_{\mathbf{c}}^{(1)}(0) = f_i(c_1, \dots, c_{i-1}, 0, c_{i+1}, \dots, c_n) = p^i = f_{\mathbf{c}}^{(2)}(0)$. Otherwise, $f_{\mathbf{c}}^{(1)}(0) = f_i(c_1, \dots, c_{i-1}, 0, c_{i+1}, \dots, c_n) = \lfloor c_{i+1}/p \rfloor = f_{\mathbf{c}}^{(2)}(0)$.

Case 2: $i = n$. If $f_{\mathbf{c}}(0) = f_n(c_1, \dots, c_{n-1}, 0)$ is determined via Case 2.a, then we have $f_{\mathbf{c}}^{(1)}(0) = f_n(c_1, \dots, c_{n-1}, 0) = p^n = f_{\mathbf{c}}^{(2)}(0)$.

Case 2.b cannot apply, since this would require $c_n \geq (c_{n-1} + 1) \cdot p$, which is impossible in case of $c_n = 0$.

Case 2.c cannot apply either, since it would require $c_n > c_{n-1} \cdot p$, which is impossible in case of $c_n = 0$.

If Case 2.d applies, then we have $f_{\mathbf{c}}^{(1)}(0) = f_n(c_1, \dots, c_{n-1}, 0) = c_{n-1} \cdot p$. If we now apply f again, then we are still in Case 2.d, i.e., $f_{\mathbf{c}}^{(2)}(0) = c_{n-1} \cdot p = f_{\mathbf{c}}^{(1)}(0)$.

Finally, if Case 2.e applies, then $f_{\mathbf{c}}^{(1)}(0) = f_n(c_1, \dots, c_{n-1}, 0) = c_{n-1} \cdot p + 1$ holds. By the definition of f_n, the last component cannot be incremented beyond $c_{n-1} \cdot p + p$ in this case. Hence, we indeed have $f_{\mathbf{c}}^{(p)}(0) = f_{\mathbf{c}}^{(p+1)}(0)$. □

B PROOF OF LEMMA 5.6

The proof follows the same line of reasoning as [19]. We use the same definition of the height of a parse tree T, namely, the number of edges on the longest path from the root to a leaf: for example, trees in Figure 3 have heights 1, 2, and 2, respectively. As usual, the internal nodes of a parse tree are labeled with non-terminals, and the leaves with terminals. Unlike [19], we do not need to label the internal nodes with pairs (x_i, j) where x_i is the non-terminal (variable) and j represents a monomial number in f_i, but we label them only with the variable x_i. This is sufficient because our terminal symbols $a_{\mathrm{v},i}$ are in one-to-one correspondence with the monomials in the polynomial f_i.

Recall that each component $(f^{(q)}(0))_i$ of $\mathbf{f}^{(q)}(0)$ is a multivariate polynomial in the coefficients $\mathbf{a} = (a_1, \dots, a_M)$ that occur in all polynomials $f_1, \dots, f_N$. When we repeatedly apply $\mathbf{f}$, each monomial $a_1^{v_1} \cdots a_M^{v_M}$ may appear multiple times in $\mathbf{f}^{(q)}(0)$; in our Example 5.5 the monomial a^3b^4 appears 1 times in $f^{(3)}(0)$, then 5 times in $f^{(4)}(0)$. Let $M_i^{(q)}$ be the bag of monomials occurring in $(f^{(q)}(0))_i$, and let $N_i^{(q)}$ be the bag of yields of all derivation trees of depth $\leq q$, i.e. $\{\{Y(T) \mid T \in \mathcal{T}_q^i\}\}$. We prove by induction on q that $M_i^{(q)} = N_i^{(q)}$ for all $i = 1, N$.

When $q = 0$ then both bags are empty. Assuming the statement holds for $q \geq 0$, we prove it for $q + 1$. We prove that $M_i^{(q+1)} \subseteq N_i^{(q+1)}$; the other direction is similar and omitted. By definition, $(\mathbf{f}^{(q+1)}(0))_i = f_i(\mathbf{f}^{(q)}(0))$, where f_i is a sum of monomials, see Equation (37). Each monomial in f_i has the form $a_j x_1^{\ell_1} \cdots x_N^{\ell_N}$ for some $a_j \in \mathbf{a}$ and some vector of exponents $(\ell_1, \ldots, \ell_N)$. Then, $f_i(\mathbf{f}^{(q)}(0))$ is a sum of expressions of the form:

$$a_j \cdot (\mathbf{f}^{(q)}(0))_1^{\ell_1} \cdots (\mathbf{f}^{(q)}(0))_N^{\ell_N} \tag{68}$$

In other words, we have substituted each x_j in (37) with $(\mathbf{f}^{(q)}(0))_j$, which, in turn, is a sum of the monomials in the bag $M_j^{(q)}$. Thus, after expanding the expression (68), we obtain a sum of monomials m of the following form:

$$m = a_j \cdot m_{1,1} \cdots m_{1,\ell_1} \cdot \cdots \cdot m_{N,1} \cdots m_{N,\ell_N} \tag{69}$$

where $m_{j,k} \in M_j^{(q)}$. To summarize, each monomial m occurring in $M_i^{(q+1)}$ can be uniquely obtained as follows: (a) choose one monomial $a_j x_1^{\ell_1} \cdots x_N^{\ell_N}$ in the polynomial f_i, (b) for each $j = 1, N$, and each $k = 1, \ell_k$, choose one monomial $m_{j,k}$ in the bag $M_j^{(q)}$ (a total of $\sum_j \ell_j$ choices). Given these choices, we define a unique[10] parse tree of depth $q + 1$ whose yield is m, as follows: (a′) The root node is labeled with the production (38) associated to the monomial that we used in (a). (b′) For each $j = 1, N$ and each $k = 1, \ell_j$, we use the inductive hypothesis $M_j^{(q)} = N_j^{(q)}$ and argue that the monomial $m_{j,k} \in M_j^{(q)}$ is in $N_j^{(q)}$, thus it is the yield of some parse tree $T_{j,k}$ of depth $\leq q$. Then, we complete our parse tree by setting $T_{j,k}$ to be the (j,k)th child of the root node. This tree generates the following word:

$$a_j \cdot m_{1,1} \cdots m_{1,\ell_1} \cdot \cdots \cdot m_{N,1} \cdots m_{N,\ell_N}$$

This is equal to m in Equation (69), proving that $m \in N_i^{(q+1)}$ and, thus, $M_i^{(q+1)} \subseteq N_i^{(q+1)}$. Containment in the other direction is similar and omitted.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 265–283. Retrieved from: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/ abadi

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

[3] Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. 2022. Convergence of Datalog over (pre-) semirings. In *International Conference on Management of Data (PODS'22)*, Leonid Libkin and Pablo Barceló (Eds.). ACM, 105–117. DOI: https://doi.org/10.1145/3517804.3524140

[4] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. 2016. FAQ: Questions asked frequently. In *35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, Tova Milo and Wang-Chiew Tan (Eds.). ACM, 13–28. DOI: https://doi.org/10.1145/2902251.2902280

[10] "Unique" means that different choices (a) and (b) will lead to different parse trees.

[5] Srinivas M. Aji and Robert J. McEliece. 2000. The generalized distributive law. *IEEE Trans. Inf. Theor* 46, 2 (2000), 325–343. DOI: https://doi.org/10.1109/18.825794

[6] Michael Arntzenius and Neel Krishnaswami. 2020. Seminaïve evaluation for a higher-order functional language. *Proc. ACM Program. Lang.* 4, POPL (2020), 22:1–22:28. DOI: https://doi.org/10.1145/3371090

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

[8] Sophie Brinke, Erich Grädel, and Lovro Mrkonjic. 2023. Ehrenfeucht-Fraïssé games in semiring semantics. *CoRR* abs/2308.04910 (2023).

[9] Bernard Carré. 1979. *Graphs and Networks.* The Clarendon Press, Oxford University Press, New York. xvi+277 pages.

[10] E. F. Codd. 1970. A relational model of data for large shared data banks. *Commun. ACM* 13, 6 (1970), 377–387. DOI: https://doi.org/10.1145/362384.362685

[11] Tyson Condie, Ariyam Das, Matteo Interlandi, Alexander Shkapsky, Mohan Yang, and Carlo Zaniolo. 2018. Scaling-up reasoning and advanced analytics on BigData. *Theor. Pract. Log. Program.* 18, 5–6 (2018), 806–845. DOI: https://doi.org/10.1017/S1471068418000418

[12] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. DOI: https://doi.org/10.1145/512950.512973

[13] Patrick Cousot and Radhia Cousot. 1992. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *Programming Language Implementation and Logic Programming (Leuven, 1992)* (Lecture Notes in Computer Science, Vol. 631). Springer, Berlin, 269–295. DOI: https://doi.org/10.1007/3-540-55844-6_142

[14] Nadia Creignou, Phokion G. Kolaitis, and Heribert Vollmer (Eds.). 2008. *Complexity of Constraints—An Overview of Current Research Themes.* (Lecture Notes in Computer Science, Vol. 5250). Springer. DOI: https://doi.org/10.1007/978-3-540-92800-3

[15] Katrin M. Dannert, Erich Grädel, Matthias Naaf, and Val Tannen. 2021. Semiring provenance for fixed-point logic. In *29th EACSL Annual Conference on Computer Science Logic (CSL'21) (LIPIcs, Vol. 183)*, Christel Baier and Jean Goubault-Larrecq (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 17:1–17:22. DOI: https://doi.org/10.4230/LIPIcs.CSL.2021.17

[16] Ariyam Das, Youfu Li, Jin Wang, Mingda Li, and Carlo Zaniolo. 2019. BigData applications from graph analytics to machine learning by aggregates in recursion. In *35th International Conference on Logic Programming (Technical Communications) (ICLP'19)*(EPTCS, Vol. 306), Bart Bogaerts, Esra Erdem, Paul Fodor, Andrea Formisano, Giovambattista Ianni, Daniela Inclezan, Germán Vidal, Alicia Villanueva, Marina De Vos, and Fangkai Yang (Eds.). 273–279. DOI: https://doi.org/10.4204/EPTCS.306.32

[17] Brian A. Davey and Hilary A. Priestley. 1990. *Introduction to Lattices and Order.* Cambridge University Press, Cambridge. Retrieved from: http://www.worldcat.org/search?qt=worldcat_org_all&q=0521367662

[18] Rina Dechter. 1997. Bucket elimination: A unifying framework for processing hard and soft constraints. *Constraints Int. J.* 2, 1 (1997), 51–55. DOI: https://doi.org/10.1023/A:1009796922698

[19] Javier Esparza, Stefan Kiefer, and Michael Luttenberger. 2010. Newtonian program analysis. *J. ACM* 57, 6 (2010), 33:1–33:47. DOI: https://doi.org/10.1145/1857914.1857917

[20] Melvin Fitting. 1985. A Kripke-Kleene semantics for logic programs. *J. Log. Program.* 2, 4 (1985), 295–312. DOI: https://doi.org/10.1016/S0743-1066(85)80005-4

[21] Melvin Fitting. 1991. Bilattices and the semantics of logic programming. *J. Log. Program.* 11, 1&2 (1991), 91–116. DOI: https://doi.org/10.1016/0743-1066(91)90014-G

[22] Melvin Fitting. 1991. Kleene's logic, generalized. *J. Log. Comput.* 1, 6 (1991), 797–810. DOI: https://doi.org/10.1093/logcom/1.6.797

[23] Melvin Fitting. 1993. The family of stable models. *J. Log. Program.* 17, 2/3&4 (1993), 197–225. DOI: https://doi.org/10.1016/0743-1066(93)90031-B

[24] Melvin Fitting. 2002. Fixpoint semantics for logic programming a survey. *Theor. Comput. Sci.* 278, 1–2 (2002), 25–51. DOI: https://doi.org/10.1016/S0304-3975(00)00330-3

[25] Robert W. Floyd. 1962. Algorithm 97: Shortest path. *Commun. ACM* 5, 6 (1962), 345. DOI: https://doi.org/10.1145/367766.368168

[26] Sumit Ganguly, Sergio Greco, and Carlo Zaniolo. 1991. Minimum and maximum predicates in logic programming. In *10th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Daniel J. Rosenkrantz (Ed.). ACM Press, 154–163. DOI: https://doi.org/10.1145/113413.113427

[27] Sumit Ganguly, Sergio Greco, and Carlo Zaniolo. 1995. Extrema predicates in deductive databases. *J. Comput. Syst. Sci.* 51, 2 (1995), 244–259. DOI: https://doi.org/10.1006/jcss.1995.1064

[28] Allen Van Gelder. 1989. The alternating fixpoint of logic programs with negation. In *8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Avi Silberschatz (Ed.). ACM Press, 1–10. DOI: https://doi.org/10.1145/73721.73722

[29] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. 1991. The well-founded semantics for general logic programs. *J. ACM* 38, 3 (1991), 620–650. DOI: https://doi.org/10.1145/116825.116838

[30] Michael Gelfond and Vladimir Lifschitz. 1988. The stable model semantics for logic programming. In *5th International Conference and Symposium on Logic Programming*, Robert A. Kowalski and Kenneth A. Bowen (Eds.). MIT Press, 1070–1080.

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

[32] Michel Gondran. 1979. Les elements p-reguliers dans les dioïdes. *Discret. Math.* 25, 1 (1979), 33–39. DOI: https://doi.org/10.1016/0012-365X(79)90150-X

[33] Michel Gondran and Michel Minoux. 2008. *Graphs, Dioids and Semirings(Operations Research/Computer Science Interfaces Series*, Vol. 41). Springer, New York, xx+383 pages.

[34] Sergio Greco, Domenico Saccà, and Carlo Zaniolo. 1995. DATALOG queries with stratified negation and choice: From P to D$^{\mathrm{P}}$. In *5th International Conference on Database Theory (ICDT'95) (Lecture Notes in Computer Science*, Vol. 893), Georg Gottlob and Moshe Y. Vardi (Eds.). Springer, 82–96. DOI: https://doi.org/10.1007/3-540-58907-4_8

[35] Sergio Greco and Carlo Zaniolo. 2001. Greedy algorithms in Datalog. *Theory Pract. Log. Program.* 1, 4 (2001), 381–407. DOI: https://doi.org/10.1017/S1471068401001090

[36] Sergio Greco, Carlo Zaniolo, and Sumit Ganguly. 1992. Greedy by choice. In *11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Moshe Y. Vardi and Paris C. Kanellakis (Eds.). ACM Press, 105–113. DOI: https://doi.org/10.1145/137097.137836

[37] Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. 2013. Datalog and recursive query processing. *Found. Trends Datab.* 5, 2 (2013), 105–195. DOI: https://doi.org/10.1561/1900000017

[38] Todd J. Green, Gregory Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *26th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Leonid Libkin (Ed.). ACM, 31–40. DOI: https://doi.org/10.1145/1265530.1265535

[39] Jiaqi Gu, Yugo H. Watanabe, William A. Mazza, Alexander Shkapsky, Mohan Yang, Ling Ding, and Carlo Zaniolo. 2019. RaSQL: Greater power and performance for big data analytics with recursive-aggregate-SQL on Spark. In *International Conference on Management of Data*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 467–484. DOI: https://doi.org/10.1145/3299869.3324959

[40] Jeremy Gunawardena. 1998. An introduction to idempotency. In *Idempotency*. Publications of the Newton Institute, Vol. 11. Cambridge University Press, Cambridge, 1–49. DOI: https://doi.org/10.1017/CBO9780511662508.003

[41] Mark W. Hopkins and Dexter Kozen. 1999. Parikh's theorem in commutative Kleene algebra. In *14th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 394–401. DOI: https://doi.org/10.1109/LICS.1999.782634

[42] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. 2016. Soufflé: On synthesis of program analyzers. In *28th International Conference on Computer Aided Verification (CAV'16) (Lecture Notes in Computer Science, Vol. 9780)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 422–430. DOI: https://doi.org/10.1007/978-3-319-41540-6_23

[43] John B. Kam and Jeffrey D. Ullman. 1976. Global data flow analysis and iterative algorithms. *J. ACM* 23, 1 (1976), 158–171. DOI: https://doi.org/10.1145/321921.321938

[44] S. C. Kleene. 1956. Representation of events in nerve nets and finite automata. In *Automata Studies*. Princeton University Press, Princeton, NJ, 3–41.

[45] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. 2014. DBToaster: Higher-order delta processing for dynamic, frequently fresh views. *VLDB J.* 23, 2 (2014), 253–278. DOI: https://doi.org/10.1007/s00778-013-0348-4

[46] Jürg Kohlas. 2003. *Information Algebras—Generic Structures for Inference*. Springer.

[47] Jürg Kohlas and Nic Wilson. 2008. Semiring induced valuation algebras: Exact and approximate local computation algorithms. *Artif. Intell.* 172, 11 (2008), 1360–1399. DOI: https://doi.org/10.1016/j.artint.2008.03.003

[48] Phokion G. Kolaitis. 1991. The expressive power of stratified programs. *Inf. Comput.* 90, 1 (1991), 50–66. DOI: https://doi.org/10.1016/0890-5401(91)90059-B

[49] Phokion G. Kolaitis and Christos H. Papadimitriou. 1991. Why not negation by fixpoint?*J. Comput. Syst. Sci.* 43, 1 (1991), 125–144. DOI: https://doi.org/10.1016/0022-0000(91)90033-2

[50] Werner Kuich. 1987. The Kleene and the Parikh theorem in complete semirings. In *Automata, Languages and Programming*. (Lecture Notes in Computer Science, Vol. 267). Springer, Berlin, 212–225. DOI: https://doi.org/10.1007/3-540-18088-5_17

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

[52] Daniel J. Lehmann. 1977. Algebraic structures for transitive closure. *Theor. Comput. Sci.* 4, 1 (1977), 59–76. DOI: https://doi.org/10.1016/0304-3975(77)90056-1

[53] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. 2006. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.* 7, 3 (2006), 499–562. DOI: https://doi.org/10.1145/1149114.1149117

[54] Leonid Libkin. 2004. *Elements of Finite Model Theory*. Springer. DOI: https://doi.org/10.1007/978-3-662-07003-1

[55] Richard J. Lipton, Donald J. Rose, and Robert Endre Tarjan. 1979. Generalized nested dissection. *SIAM J. Numer. Anal.* 16, 2 (1979), 346–358. DOI: https://doi.org/10.1137/0716027

[56] Richard J. Lipton and Robert Endre Tarjan. 1980. Applications of a planar separator theorem. *SIAM J. Comput.* 9, 3 (1980), 615–627. DOI: https://doi.org/10.1137/0209046

[57] Yanhong A. Liu and Scott D. Stoller. 2020. Founded semantics and constraint semantics of logic rules. *J. Log. Comput.* 30, 8 (2020), 1609–1668. DOI: https://doi.org/10.1093/logcom/exaa056

[58] Yanhong A. Liu and Scott D. Stoller. 2022. Recursive rules with aggregation: A simple unified semantics. *J. Log. Comput.* 32, 8 (2022), 1659–1693. DOI: https://doi.org/10.1093/LOGCOM/EXAC072

[59] Michael Luttenberger and Maximilian Schlund. 2016. Convergence of Newton's method over commutative semirings. *Inf. Comput.* 246 (2016), 43–61. DOI: https://doi.org/10.1016/j.ic.2015.11.008

[60] David Maier, K. Tuncay Tekle, Michael Kifer, and David Scott Warren. 2018. Datalog: Concepts, History, and Outlook. In *Declarative Logic Programming: Theory, Systems, and Applications*, Michael Kifer and Yanhong Annie Liu (Eds.). ACM / Morgan & Claypool, 3–100. DOI: https://doi.org/10.1145/3191315.3191317

[61] Mirjana Mazuran, Edoardo Serra, and Carlo Zaniolo. 2013. Extending the power of datalog recursion. *VLDB J.* 22, 4 (2013), 471–493. DOI: https://doi.org/10.1007/s00778-012-0299-1

[62] Frank McSherry. 2022. Recursion in Materialize. Retrieved from https://github.com/frankmcsherry/blog/blob/master/posts/2022-12-25.md

[63] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. Springer-Verlag, Berlin. xxii+450 pages. DOI: https://doi.org/10.1007/978-3-662-03811-6

[64] Jorge Nocedal and Stephen J. Wright. 1999. *Numerical Optimization*. Springer. DOI: https://doi.org/10.1007/b98874

[65] Rohit J. Parikh. 1966. On context-free languages. *J. Assoc. Comput. Mach.* 13 (1966), 570–581. DOI: https://doi.org/10.1145/321356.321364

[66] D. L. Pilling. 1973. Commutative regular equations and Parikh's theorem. *J. London Math. Soc. (2)* 6 (1973), 663–666. DOI: https://doi.org/10.1112/jlms/s2-6.4.663

[67] Teodor C. Przymusinski. 1989. Every logic program has a natural stratification and an iterated least fixed point model. In *8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Avi Silberschatz (Ed.). ACM Press, 11–21. DOI: https://doi.org/10.1145/73721.73723

[68] Teodor C. Przymusinski. 1990. The well-founded semantics coincides with the three-valued stable semantics. *Fundam. Inform.* 13, 4 (1990), 445–463.

[69] Thomas W. Reps, Emma Turetsky, and Prathmesh Prabhu. 2016. Newtonian program analysis via tensor product. In *43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16)*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 663–677. DOI: https://doi.org/10.1145/2837614.2837659

[70] Tim Rocktäschel. 2018. Einsum is all you need—Einstein summation in deep learning. Retrieved from https://rockt.github.io/2018/04/30/einsum

[71] Kenneth A. Ross and Yehoshua Sagiv. 1992. Monotonic aggregation in deductive databases. In *11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Moshe Y. Vardi and Paris C. Kanellakis (Eds.). ACM Press, 114–126. DOI: https://doi.org/10.1145/137097.137852

[72] Günter Rote. 1990. Path problems in graphs. In *Computational Graph Theory*(*Comput. Suppl.*, Vol. 7). Springer, Vienna, 155–189. DOI: https://doi.org/10.1007/978-3-7091-9076-0_9

[73] Prakash P. Shenoy and Glenn Shafer. 1988. Axioms for probability and belief-function proagation. In *4th Annual Conference on Uncertainty in Artificial Intelligence (UAI'88)*, Ross D. Shachter, Tod S. Levitt, Laveen N. Kanal, and John F. Lemmer (Eds.). North-Holland, 169–198.

[74] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big data analytics with Datalog queries on Spark. In *International SIGMOD Conference on Management of Data*. 1135–1149. DOI: https://doi.org/10.1145/2882903.2915229

[75] Alexander Shkapsky, Mohan Yang, and Carlo Zaniolo. 2015. Optimizing recursive queries with monotonic aggregates in DeALS. In *31st IEEE International Conference on Data Engineering (ICDE'15)*, Johannes Gehrke, Wolfgang Lehner, Kyuseok Shim, Sang Kyun Cha, and Guy M. Lohman (Eds.). IEEE Computer Society, 867–878. DOI: https://doi.org/10.1109/ICDE.2015.7113340

[76] Richard P. Stanley. 1999. *Enumerative Combinatorics. Vol. 2* (Cambridge Studies in Advanced Mathematics, Vol. 62). Cambridge University Press, Cambridge. xii+581 pages. DOI: https://doi.org/10.1017/CBO9780511609589

[77] Robert E. Tarjan. 1976. Graph theory and Gaussian elimination. J. R. Bunch and D. J. Rose (Eds.). 3–22.

[78] Robert Endre Tarjan. 1981. A unified approach to path problems. *J. ACM* 28, 3 (1981), 577–593. DOI: https://doi.org/10.1145/322261.322272

[79] Moshe Y. Vardi. 1982. The complexity of relational query languages (extended abstract). In *14th Annual ACM Symposium on Theory of Computing*, Harry R. Lewis, Barbara B. Simons, Walter A. Burkhard, and Lawrence H. Landweber (Eds.). ACM, 137–146. DOI: https://doi.org/10.1145/800070.802186

[80] Victor Vianu. 2021. Datalog unchained. In *40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS'21)*, Leonid Libkin, Reinhard Pichler, and Paolo Guagliardo (Eds.). ACM, 57–69. DOI: https://doi.org/10.1145/3452021.3458815

[81] Yisu Remy Wang, Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, and Dan Suciu. 2022. Optimizing recursive queries with progam synthesis. In *International SIGMOD Conference on Management of Data*, Zachary Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 79–93. DOI: https://doi.org/10.1145/3514221.3517827

[82] Stephen Warshall. 1962. A theorem on Boolean matrices. *J. ACM* 9, 1 (1962), 11–12. DOI: https://doi.org/10.1145/321105.321107

[83] Carlo Zaniolo, Ariyam Das, Jiaqi Gu, Youfu Li, Mingda Li, and Jin Wang. 2019. Monotonic properties of completed aggregates in recursive queries. *CoRR* abs/1910.08888 (2019).

[84] Carlo Zaniolo, Ariyam Das, Jiaqi Gu, Youfu Li, Mingda Li, and Jin Wang. 2021. Developing big-data Application as queries: An aggregate-based approach. *IEEE Data Eng. Bull.* 44, 2 (2021), 3–13. Retrieved from: http://sites.computer.org/debull/A21june/p3.pdf

[85] Carlo Zaniolo, Mohan Yang, Ariyam Das, and Matteo Interlandi. 2016. The magic of pushing extrema into recursion: Simple, powerful Datalog programs. In *10th Alberto Mendelzon International Workshop on Foundations of Data Management (CEUR Workshop Proceedings*, Vol. 1644), Reinhard Pichler and Altigran Soares da Silva (Eds.). CEUR-WS.org. Retrieved from: http://ceur-ws.org/Vol-1644/paper16.pdf

[86] Carlo Zaniolo, Mohan Yang, Ariyam Das, Alexander Shkapsky, Tyson Condie, and Matteo Interlandi. 2017. Fixpoint semantics and optimization of recursive Datalog programs with aggregates. *Theory Pract. Log. Program.* 17, 5–6 (2017), 1048–1065. DOI: https://doi.org/10.1017/S1471068417000436

[87] Carlo Zaniolo, Mohan Yang, Matteo Interlandi, Ariyam Das, Alexander Shkapsky, and Tyson Condie. 2018. Declarative BigData algorithms via aggregates and relational database dependencies. In *12th Alberto Mendelzon International Workshop on Foundations of Data Management*(CEUR Workshop Proceedings, Vol. 2100), Dan Olteanu and Barbara Poblete (Eds.). CEUR-WS.org. Retrieved from: http://ceur-ws.org/Vol-2100/paper2.pdf

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

Received 1 February 2023; revised 22 December 2023; accepted 9 January 2024