University of Nebraska - Lincoln

DigitalCommons@University of Nebraska - Lincoln

CSE Conference and Workshop Papers

Computer Science and Engineering, Department of

9-10-1995

Safe stratified Datalog with integer order programs

Peter Revesz

University of Nebraska-Lincoln, prevesz1@unl.edu

Follow this and additional works at: <http://digitalcommons.unl.edu/cseconfwork>

Part of the [Computer Engineering Commons](#), [Electrical and Computer Engineering Commons](#), and the [Other Computer Sciences Commons](#)

Revesz, Peter, "Safe stratified Datalog with integer order programs" (1995). *CSE Conference and Workshop Papers*. 321.
<http://digitalcommons.unl.edu/cseconfwork/321>

This Article is brought to you for free and open access by the Computer Science and Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in CSE Conference and Workshop Papers by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.

Safe Stratified Datalog with Integer Order Programs

Peter Z. Revesz

Department of Computer Science and Engineering
and Center for Communication and Information Science
University of Nebraska–Lincoln, Lincoln, NE 68588, USA

Abstract. Guaranteeing termination of programs on all valid inputs is important for database applications. Termination cannot be guaranteed in Stratified Datalog with integer (gap)-order, or $Datalog^{\neg, < \mathbf{z}}$, programs on generalized databases because they can express any Turing-computable function [23]. This paper introduces a restriction of $Datalog^{\neg, < \mathbf{z}}$ that can express only computable queries. The restricted language has a high expressive power and a non-elementary data complexity.

1 Introduction

Constraint logic programming [14, 15, 27, 12, 10, 9] has a great potential for being adapted for database use. A successful adaptation of constraint logic programming has to meet usual database requirements. In the constraint query languages framework [19] two requirements are identified as especially important: (a) closed-form evaluation and (b) bottom-up processing.

Closed-form evaluation means that all possible tuple answers to a query are represented finitely by an output constraint database that has the same type of constraints as the input constraint database. This is the analogue for the relational database model requirement that relations be finite structures and queries preserve finiteness. The main advantage of closed-form evaluation is that it facilitates composition of queries. In particular, a query may be applied to the *negation* of the output of another query in a natural way. This composition is called *stratified negation* [6, 2] (see Section 2 for further discussion). Closed-form evaluation also allows the addition of aggregate operators as is done recently in [8]. Aggregation is also important in database applications.

Bottom-up processing describes a direction of evaluation of rules within programs. The direction is from known facts to goals. Bottom-up processing can be done in a set-at-a-time way which is faster than the tuple-at-a-time top-down processing done in Prolog. Bottom-up processing requires less access to secondary storage [20] and allows algebraic query optimizations [4, 18].

The good news is that these requirements can be met in a large number of cases. For example, Datalog with rational order constraints can be evaluated bottom-up in closed-form in PTIME data complexity. (Data complexity is the measure of the computational complexity of fixed queries as the size of the input database grows [5, 29]. The rationale behind this measure is that in practice the size of the database typically dominates by several orders of magnitude

the size of the query.) Datalog with integer (gap)-order constraints programs and Datalog with $\subseteq$ constraints on set variables are evaluable in closed-form on constraint databases with DEXPTIME-complete data complexity [23, 11, 24]. Datalog_{IS} [7], an extension of Datalog with a successor function applied always to the first argument of relations, can be evaluated in closed-form and has PSPACE-complete data complexity. Datalog with periodicity constraints [26], relational calculus with linear repeating points [17] and temporal constraints [17] can be also evaluated in closed-form.

The bad news is that many other interesting constraint logic programming languages do not guarantee a closed-form evaluation. For example, CLP(R) [16], LIFE [1], the temporal database queries of [3], and stratified Datalog with integer (gap)-order constraints [23] can express any Turing-computable function, hence in these languages termination of query evaluation cannot be guaranteed. In spite of their high expressive power, using these languages for database applications can be difficult.

Developers can translate faster from user specifications to a declarative language than to a procedural language. This is a major advantage of using relational database systems.¹ However, in practice, developers not only have to express the desire of the users as queries, but they also must guarantee that those queries terminate on each possible input. (Obviously, this is important for user satisfaction.) While translating may be easy, guaranteeing termination may be difficult in the above languages. The time saved in translation evaporates rapidly if developers have to prove termination of programs.

It seems ideal if a developer can work in a highly expressive language where however termination is guaranteed. Therefore it seems important to look at restricted cases of the above languages. In this paper, we consider a syntactical restriction of stratified $Datalog^{\neg, <^z}$ programs. Queries expressible in this restricted language are called *safe*. Safe stratified $Datalog^{\neg, <^z}$ queries are guaranteed to be evaluable bottom-up in closed-form on any valid database input. The syntactical restriction is easy to check: it can be done in PTIME in the size of the $Datalog^{\neg, <^z}$ programs. Moreover, this restriction still leaves developers with a highly expressive language: safe stratified $Datalog^{\neg, <^z}$ queries have a non-elementary data complexity.

Section 2 describes basic definitions. Section 3 gives a definition of safety and shows that it can be checked in PTIME in the size of stratified $Datalog^{\neg, <^z}$ programs. Section 4 presents an evaluation algorithm for safe stratified $Datalog^{\neg, <^z}$ queries. It is shown that the evaluation algorithm always terminates in finite time and returns a constraint database representation of the perfect fixpoint model of the query. Section 5 analyzes the computational complexity of safe stratified $Datalog^{\neg, <^z}$ queries. The complexity of these queries is shown to be non-elementary. It is also shown that the level of exponentiation can grow linearly with the number of strata in the programs.

¹ In fact, relational database systems also allow so called ad hoc querying by naive users.

2 Basic Concepts

2.1 Definition of Syntax and Semantics

We denote sets of rules by capital letter R and individual rules (predicates) by small case letter r (p) with or without subscripts. We also use small case letters for integer variables.

The syntax of Datalog with integer (gap)-order programs, denoted $Datalog^{<^z}$, is that of traditional Datalog (Horn clauses without function symbols) where the bodies of rules can also contain a conjunction of integer (gap)-order constraints. That is, each program is a finite set of rules of form: $A_0 \text{ :- } A_1, A_2, \dots, A_l$. The expression A_0 (the rule *head*) must be an atomic formula of the form $p(x_1, \dots, x_n)$, and the expressions $A_1, \dots, A_l$ (the rule *body*) must be atomic formulas of the form $x_i = x_j$, $x_i \neq x_j$, $x_i \leq x_j$, $x_i < x_j$, $x_i <_g x_j$ where g is a nonnegative integer, or $p(x_1, \dots, x_n)$, where p is some predicate symbol.

The definitions below generalize those in [28] to $Datalog^{<^z}$ programs and finitely representable relations.

The rules for a predicate p are *rectified* if all their heads are identical and of the form $p(x_1, \dots, x_k)$ for distinct variables $x_1, \dots, x_k$.

We call *rule instantiation* a substitution of each variable of a rule by an integer constant.

Let $p_1, \dots, p_n$ be any set of relation symbols in a language. An *assignment* is a (possibly infinite) set of tuples of proper arity for each p_i where $1 \leq i \leq n$. For any assignment A and rule instantiation, the right hand side is “true” if and only if the argument list of each instantiated subgoal with a relation symbol p appears as a tuple for p in A , and all the instantiated constraints are also satisfied.

Let R be a set of $Datalog^{<^z}$ rules. Let $p_1, \dots, p_n$ be the relation symbols in R that occur only on the right hand sides. We call each assignment to $p_1, \dots, p_n$ an *extensional database* (EDB) of R and each p_i an EDB relation symbol.

Let R be a set of $Datalog^{<^z}$ rules. Let $p_1, \dots, p_m$ be the EDB relation symbols in R , and let $p_{m+1}, \dots, p_n$ be the IDB relation symbols (i.e, those relation symbols that occur at least once on the left hand sides). A *model* of R with respect to an extensional database E of R is an assignment M to $p_1, \dots, p_n$ such that $M = E \cup I$ where I is an assignment to $p_{m+1}, \dots, p_n$, and the following holds for each $r_i \in R$:

For each instantiation $\sigma = \{x_1 = a_1, \dots, x_t = a_t\}$ of r_i , where $x_1, \dots, x_t$ are the variables in r_i , if the right hand side is true, then the left hand side is also true.

Let R be a set of rectified $Datalog^{<^z}$ rules with $p_1, \dots, p_m$ EDB relation symbols and $p_{m+1}, \dots, p_n$ IDB relation symbols. Let E be an extensional database of R . A *fixpoint model* of R with respect to E is a model M of R with respect to E , such that for each tuple $(a_1, \dots, a_k)$ for p_j in M where $m + 1 \leq j \leq n$, the following holds:

There is a rule $r_i \in R$ with head $p_j(x_1, \dots, x_k)$, variables $x_1, \dots, x_k, \dots, x_t$, and an instantiation $\sigma = \{x_1 = a_1, \dots, x_t = a_t\}$ such that the right hand side is

true.

Let R be a set of $Datalog^{<Z}$ rules and E be an extensional database of R . We say that F is a *least fixpoint model* of R with respect to E , if F is a fixpoint model of R with respect to E , and there is no $F' \subset F$ such that F' is also a fixpoint model of R with respect to E .

The syntax of Stratified Datalog with integer (gap)-order programs, denoted stratified $Datalog^{\neg, <Z}$, is that of $Datalog^{<Z}$ except in the rule bodies expressions of the form $\neg p(x_1, \dots, x_n)$ can also occur.

A stratification of a program means the grouping of defined predicates (and the rules defining them) into a set of disjoint subgroups in order $R_1, \dots, R_n$. A stratification is *correct* if for each rule of the form $p :- \dots \neg q \dots$, the predicate p has a higher group number than q has. The intuition here is that during fixpoint computation the lower strata have to be fully evaluated before the higher strata. Each stratified $Datalog^{\neg, <Z}$ query is prescribed a unique meaning in terms of a perfect model.

Let $R = R_1 \cup \dots \cup R_n$ be a set of stratified rules, where stratum i contains rules R_i . Let M_0 be an assignment to the EDB relation symbols in R . The *perfect fixpoint model* of R with respect to M_0 is $M = M_0 \cup I_1 \cup \dots \cup I_n$ where each I_i is an assignment to the IDB relation symbols in R_i , and for each $1 \leq i \leq n$ the assignment $M_i = M_{i-1} \cup I_i$ is a least fixpoint model of R_i with respect to M_{i-1} . (We also amend the previous definitions by assuming that if $\neg p(a_1, \dots, a_k)$ is an instantiated predicate on the right hand side, then it is true if and only if $(a_1, \dots, a_k)$ is not assigned to p .)

In this paper we are not concerned about testing whether a given stratification is correct, or whether a $Datalog^{\neg, <Z}$ program can have a correct stratification and how to find a correct stratification if one exists. These questions can be answered by algorithms given in [28]. We will always assume that for each program we already have a correct stratification.

2.2 Definition of Technical Tools

The following definitions and lemmas are either given in or trivially follow from [23].

Definition 2.1 Let x and y be any two integer variables or constants. Given some assignment to the variables, a *gap-order* constraint $x <_g y$ for some gap-value $g \in \mathbb{N}$ holds if and only if $g < y - x$ holds in the given assignment. A *gap-order* constraint $x = y$ holds if and only if x and y are equal in the given assignment. $\square$

Definition 2.2 Let $x_1, \dots, x_n$ be integer variables and $c_1, \dots, c_m$ be integer constants. Any graph with vertices labeled $x_1, \dots, x_n, c_1, \dots, c_m$ and at most one undirected edge labeled by $=$ or at most one directed edge labeled by $<_g$ for some $g \in \mathbb{N}$ between any pair of distinct vertices is called a *gap-graph*. $\square$

Remark: It should be clear that each gap-graph represents a set of gap-order constraints. In the case of directed edges the left hand side of the gap-order constraint is the vertex of origin and the right hand side of the gap-order

constraint is the vertex of incidence of the directed edge. It is immediate that gap-graphs can represent any set of gap-order constraints S over variables X and constants C if between any two $v, u \in X \cup C$ at most one of $v = u$ or $v <_g u$ for some $g \in \mathbf{N}$ is in S . It is also transparent that gap-graphs can represent any set of gap-order constraints, and disjunctions of gap-graphs can represent any set of $=, \neq, \leq, \geq, <, >, <_g$ constraints.

Definition 2.3 Let $x_1, \dots, x_n$ be integer variables and l, u be integer constants. Any gap-graph with vertices labeled $x_1, \dots, x_n, l, u$ is in *(l,u)-standard form*. Furthermore, any set of gap-graphs each with vertices labeled $x_1, \dots, x_n, l, u$ is in *(l,u)-standard form*. $\square$

Definition 2.4 We say that two gap-graphs G_1 and G_2 are *equivalent* if and only if the following two conditions are both satisfied:

- (1) G_1 and G_2 have the same set of non-constant labelled vertices $x_1, \dots, x_n$.
- (2) Any assignment of integers to $x_1, \dots, x_n$ satisfies the gap-order constraints in G_1 if and only if it satisfies the gap-order constraints in G_2 . $\square$

Remark: We use $\mathcal{A}(G)$ to denote all the assignments that satisfy G . Then an alternative way of stating that G_1 and G_2 are equivalent is to say that $\mathcal{A}(G_1) = \mathcal{A}(G_2)$.

Lemma 2.1 Let G be any gap-graph with smallest constant vertex label $\geq l$ and largest constant vertex label $\leq u$. Then G can be put into an equivalent (l,u)-standard form. Furthermore, let S be any set of gap-graphs each with a (possibly different) smallest constant vertex label $\geq l$ and largest constant vertex label $\leq u$. Then S can be put into an equivalent (l,u)-standard form. $\square$

We say that a relation r with arity k containing exactly a (finite or infinite) set of tuples $\mathcal{B}$ is representable in gap-graph form if there are $l, u \in \mathbf{N}$ and a finite set of gap-graphs $G_1, \dots, G_n$ each over the vertices $x_1, \dots, x_k, l, u$ such that $\mathcal{B} = \bigcup_{1 \leq i \leq n} \mathcal{A}(G_i)$. We say that a database is representable in gap-graph form if each of the relations in it is representable in gap-graph form.

The main motivation for using gap-graphs is that least models can be computed in gap-graph form. More precisely, the following is shown in [23].

Lemma 2.2 Let P be *Datalog*^{< z} program and B be a (possibly infinite) regular relational database. If the database B is representable in gap-graph form, then $L_{P,B}$ the least model of P on B is also representable in gap-graph form. Furthermore, given any gap-graph representation of B a gap-graph representation of $L_{P,B}$ can be computed in finite time.

3 An Algorithm to Test Safety

It is traditional in the relational database literature to define various “safety restrictions” on languages to ensure that queries in the restricted language always yield finite database outputs on finite database inputs [28]. We generalize this notion of safety. Our aim is to ensure that queries in the restricted language

always yield finitely representable generalized database outputs on finitely represented generalized database inputs.

In this section we define a syntactic notion of safety that can be tested in PTIME in the size of the programs. In the next section, we show that safe $Datalog^{\neg, <^z}$ programs can be evaluated in finite time.

At first let us give an intuition to the problem of computing perfect models. Suppose that we want to evaluate the i th stratum of a stratified $Datalog^{\neg, <^z}$ program. What we need intuitively is to find the complement of the negated relations that are either fully evaluated in the previous strata or given in the input database. If we could represent in gap-graph form the complement of each negated relation occurring in the i th stratum, then we could apply Lemma 2.2 and find in finite time a least model of stratum i . The task then is to find those cases when the negated relations are always surely representable in gap-graph form.

We do that in two steps. First, we note that if a relation has a certain simple form, then its negation is representable in gap-graph form. Second, we make a type for each input relation that will tell whether it has a simple form. This will allow calculating the type of each output relation. This essentially can be considered a type checking. Using information about the program syntax, the stratification of the rules in the program, and the type of the input relations, this type checking will approve programs for which termination of evaluation can be guaranteed for any valid input database.

Definition 3.1 A gap-graph is simple if it contains no $<$ or $<_g$ constraint between any pair of variables.

Lemma 3.1 Let p be a relation represented by a set of simple gap-graphs. Then $\neg p$ can be represented as a set of simple gap-graphs. $\square$

Example 3.1 Suppose $p(x) = 20 <_5 x \vee x <_4 7$. Here $l = 7$ and $u = 20$. Then applying De Morgan's laws we have $\neg p(x) = \neg(20 <_5 x \vee x <_4 7) = \neg(25 < x \vee x < 3) = \neg(25 < x) \wedge \neg(x < 3) = (x \leq 25) \wedge (3 \leq x) = x < 26 \wedge 2 < x$. Note that in this rewriting $l = 2$ and $u = 26$. This rewriting makes $u - l$ minimal as in any other rewriting we get either a smaller l or a larger u .

Motivated by the above, we define for each relation a technical tool called a *congraph*. Intuitively, each congraph shows the possible connections via $<$ or $<_g$ constraints among the arguments of a relation. More precisely:

Definition 3.2 Let p be any k -ary predicate. Then the arguments connection graph or *congraph of p* is an assignment of an undirected graph $C(V, E)$ with $V = \{\$1, \dots, \$k\}$. (We assume that the congraph vertex $\$i$ represents the i th argument of p .)

Let $p(x_1, \dots, x_k)$ be a k -arity relation and let $C(V, E)$ be any congraph of p . We say that C *pictures* p if the following holds: $(\$i, \$j) \in E$ if there is a path not using l and u from x_i to x_j (or from x_j to x_i) in any gap-graph in p .

We define the congraph of a rule based on the congraphs of the subgoals (atomic formulas on its left hand side).

Definition 3.3 Let r be any rule with variables $x_1, \dots, x_n$ and of the form $A_0 :- A_1, A_2, \dots, A_l$. Then the *congraph* of r is the undirected graph $C(V, E)$ with the vertices labeled $x_1, \dots, x_n$ that has in it an edge between x_i and x_j if and only if some A_j for $1 \leq j \leq l$ is an integer (gap)-order constraint involving both x_i and x_j or it is of the form $p(\dots, x_i, \dots, x_j, \dots)$ and the congraph of p has an edge between $\$i$ and $\$j$ where x_i (x_j) is the $\$i$ 'th ($\$j$ 'th) argument of p .

Our definition of safety is purely syntactic and is based on an algorithm called *CheckSafety*.

Definition 3.4 We say that a program P is safe if and only if algorithm *CheckSafety* returns “yes”.

Algorithm CheckSafety

INPUT: A stratified $Datalog^{\neg, <^z}$ program P and a congraph for each EDB.

A stratification of program P is also given.

OUTPUT: “Yes” if P is safe if the congraphs picture the EDBs.

FOR each IDB relation p_m with arity k **DO**

assign to p_m a congraph $C_m(V_m, E_m)$ with $V_m = \{\$1, \dots, \$k\}$ and $E_m = \emptyset$.

END-FOR

FOR each stratum i **DO**

WHILE any changes in congraphs **DO**

FOR each rule r_j with head $p_m(x_1, \dots, x_k)$ in stratum i **DO**

Find $C_{r_j}(V_{r_j}, E_{r_j})$ the congraph of rule r_j .

Let $E_m = E_m \cup \{(\$i, \$j) : x_i \text{ and } x_j \text{ are connected in } E_{r_j}\}$.

END-FOR

END-WHILE

END-FOR

Output “yes” if in each negated IDBP congraph each vertex is an isolate vertex.

In step (1) of the inner for loop the congraph of rule r_j is found by taking after the appropriate renamings the union of all the vertices and edges in the congraphs of the relations on the right hand side of r_j .

Theorem 3.1 The safety testing algorithm runs in PTIME in the size of the stratified $Datalog^{\neg, <^z}$ program. $\square$

Next we give an example of applying algorithm *CheckSafety*.

Example 3.2 Suppose that we know the distance in miles between any pair of cities with a direct road connection on a map and we need to find the length of the shortest path between any pair of cities. The following $Datalog^{\neg, <^z}$ program with four rules, r_1, r_2, r_3, r_4 respectively, performs this query.

$$shortest(x, y, s) \quad \text{---} \quad path(x, y, 0, s), \neg not_shortest(x, y, s).$$

$$not_shortest(x, y, s_2) \text{---} path(x, y, 0, s_1), path(x, y, 0, s_2), s_1 < s_2.$$

$$path(x, y, s_1, s_2) \quad \text{---} \quad path(x, z, s_1, s_3), distance(z, y, s_3, s_2).$$

$$path(x, y, s_1, s_2) \quad \text{---} \quad distance(x, y, s_1, s_2).$$

In the program *path* and *not_shortest* are in the first and *shortest* is in the second stratum. The input relation *distance* describes direct distances between cities in miles using constraint database tuples. For example, to express the fact that city 77 is 60 miles from city 95 we use the constraint tuple:

$$distance(95, 77, s_1, s_2) \text{---} s_1 <_{59} s_2.$$

This constraint tuple should be read as follows: if we can reach city 95 within s_1 miles then we can reach city 77 within s_2 miles for any s_1 and s_2 that satisfies $s_1 <_{59} s_2$.

Figure 1 shows the edges in the congraphs of each rule and each relation at the end of each iteration i . For $i = 0$ the congraphs of the input database are shown. The input database relation *distance* will have in its congraph only the edge \$3-\$4, and all the other relations will not have any edge in their congraphs. None of the rule congraphs will have any edge in them either.

After the first iteration, the congraph of rules r_1 and r_3 will have no edges, the congraph of r_2 will have only the edge $s_1 - s_2$ because of the constraint $s_1 < s_2$ occuring in the rule, while the congraph of r_4 will have the edge $s_1 - s_2$ added to it, because on the right hand side the *distance* relation also contains this edge. Because of the change in r_4 , the congraph of *path* will also have the edge \$3-\$4 added to it.

After the second iteration, the congraph of rules r_1, r_2 and r_4 will remain unchanged, while the congraph of r_3 will have the edge $s_1 - s_2$ added to it. This change in r_3 however will not cause any change in the congraph of the path relation. Therefore, none of the relation congraphs will change from the end of iteration 1 to the end of iteration 2. Hence the algorithm will terminate and return the congraphs of the last row.

Clearly the only negated relation is *not_shortest*. Since the congraph of *not_shortest* contains no edges, the query must be a safe query. $\square$

iteration	<i>distance</i>	<i>path</i>	<i>not_shortest</i>	<i>shortest</i>	r_1	r_2	r_3	r_4
0	\$3-\$4							
1	\$3-\$4	\$3-\$4				$s_1 - s_2$		$s_1 - s_2$
2	\$3-\$4	\$3-\$4				$s_1 - s_2$	$s_1 - s_2$	$s_1 - s_2$

Fig. 1. The relation and rule congraphs after each iteration

4 An Evaluation Algorithm for Safe $Datalog^{\neg, <^z}$ Queries

In this section the operators $\bowtie$ and $\hat{\pi}$ that we use are the extensions of the relational algebra operators of join and project for generalized databases. These operators and their semantics are defined in [23] and for brevity of space we refer to that paper for examples. We also use ρ as a symbol for the renaming operator.

Algorithm EvalQuery

INPUT: A safe stratified $Datalog^{\neg, <^z}$ program P and a set of gap-graphs G_i for each p_i . For the defined relations $G_i = \emptyset$. A stratification of program P is also given.

OUTPUT: The perfect fixpoint model of P in gap-graph form.

```

FOR each stratum  $i$  DO
  REPEAT
    FOR each relation  $p_m$  DO
      Let  $H_m = G_m$ .
    END-FOR
    FOR each  $r_j$  of form  $p_0(x_1, \dots, x_k) \leftarrow p_1, \dots, p_n, \neg p_{n+1}, \dots, \neg p_{n+l}$  DO
      IF stratum[ $r_j$ ] =  $i$  THEN
         $T_j = \rho_{j,1}(G_1) \bowtie \dots \bowtie \rho_{j,n}(G_n) \bowtie (\neg(\rho_{j,n+1}(G_{n+1}))) \bowtie \dots \bowtie (\neg(\rho_{j,n+l}(G_{n+l})))$ .
         $F_j = \rho_{x_1/\$1, \dots, x_k/\$k}(\hat{\pi}_{x_1, \dots, x_k} T_j)$ .
        Delete all inconsistent gap-graphs from  $F_j$ .
        Add to  $G_0$  each gap-graph in  $F_j$  that does not subsume another in  $G_0$ .
      END-IF
    END-FOR
  UNTIL  $H_m = G_m$  for each  $m$ 
END-FOR

```

The next lemma shows that the query evaluation algorithm always returns relations with a type that conforms to our expectations based on the check safety algorithm.

Lemma 4.1 Let $p_1, \dots, p_n$ be the input relations and $p_{n+1}, \dots, p_{n+l}$ be the defined relations of a stratified $Datalog^{\neg, <^z}$ program P .

Let $c_{n+1}, \dots, c_{n+l}$ be the congraphs returned for $p_{n+1}, \dots, p_{n+l}$ by algorithm *CheckSafety* on P , the stratification of P and congraphs $c_1, \dots, c_n$ for $p_1, \dots, p_n$ respectively. Then for any assignment to $p_1, \dots, p_n$ algorithm *EvalQuery* returns $p_{n+1}, \dots, p_{n+l}$ such that if each c_i pictures p_i for $1 \leq i \leq n$, then each c_j pictures p_j for $1 \leq j \leq n$. $\square$

Next we prove that for safe programs the query evaluation algorithm returns in finite time the perfect model as expected.

Theorem 4.1 Algorithm *EvalQuery* terminates for any safe stratified *Datalog*^{¬, <_z} program *P* and valid input database *d* and returns the perfect fixpoint model of *P* in gap-graph form.

Proof. We prove the theorem by induction on the number of strata in *P*. That the first stratum terminates follows by Lemma 2.2 from [23] which shows that any program that adds only gap-graphs with a fixed *l* and *u* to the database terminates. Each time the evaluation enters a new stratum and negation is performed the values of *l* and *u* may increase but by Lemma 4.1 the complement of the negated relation can be also represented in gap-graph form. The only time *l* and *u* may increase is when computing the complement of the relations. Then the evaluation of the next layer terminates as it adds only gap-graphs with a fixed *l* and *u* to the database. This shows termination of the computation. The proof that the computation returns the perfect model is similar to the model theory proof in [23] and is omitted here.

Example 4.1 Let's return to Example 3.2. We have seen that it was identified to be a safe query. Now we show how it will be evaluated by *EvalQuery* on the following input database:

$distance(1, 2, s_1, s_2) :- s_1 <_{19} s_2.$
 $distance(1, 3, s_1, s_2) :- s_1 <_{44} s_2.$
 $distance(2, 4, s_1, s_2) :- s_1 <_{29} s_2.$
 $distance(3, 4, s_1, s_2) :- s_1 <_{14} s_2.$

Let G_R^i and F_j^i denote the set of gap-graphs assigned respectively to IDB relation *R* and to rule *r_j* at the end of iteration *i* of the repeat-until loop.

We have $G_d = \{g_1, g_2, g_3, g_4\}$ where g_1 is the gap-graph for $\$1 = 1, \$2 = 2, \$3 <_{19} \4 , g_2 is $\$1 = 1, \$2 = 3, \$3 <_{44} \4 , g_3 is $\$1 = 2, \$2 = 4, \$3 <_{29} \4 and g_4 is $\$1 = 3, \$2 = 4, \$3 <_{14} \4 . We also have $G_p^0 = G_{ns}^0 = G_s^0 = \emptyset$.

Let's see now what happens when algorithm *EvalQuery* enters stratum 1 which contains rules r_2, r_3 and r_4 . For each iteration *i* of the repeat-until loop, the algorithm finds:

$$\begin{aligned} F_2^i &= \rho_{x/\$1, y/\$2, s_2/\$3}(\hat{\pi}_{x, y, s_2}(\rho_{\$1/x, \$2/y, \$3/0, \$4/s_1} G_p^{i-1} \bowtie \\ &\quad \rho_{\$1/x, \$2/y, \$3/0, \$4/s_2} G_p^{i-1} \bowtie \rho_{\$1/s_1, \$2/s_2} G <)) \\ F_3^i &= \rho_{x/\$1, y/\$2, s_1/\$3, s_2/\$4}(\hat{\pi}_{x, y, s_1, s_2}(\rho_{\$1/x, \$2/z, \$3/s_1, \$4/s_3} G_p^{i-1} \bowtie \\ &\quad \rho_{\$1/z, \$2/y, \$3/s_3, \$4/s_2} G_d)) \\ F_4^i &= \rho_{x/\$1, y/\$2, s_1/\$3, s_2/\$4}(\hat{\pi}_{x, y, s_1, s_2}(\rho_{\$1/x, \$2/y, \$3/s_1, \$4/s_2} G_d)) \end{aligned}$$

Note that $F_4^i = G_d$ because the two renaming operators cancel each other.

In the first iteration of the repeat-until loop, we have $G_p^0 = \emptyset$. Therefore, both F_2^1 and F_3^1 will be empty. As we noted, $F_4^1 = G_d$. This has the net effect of copying each gap-graph in the distance to the path relation. Hence by the end of the first iteration, we have $G_s^1 = G_{ns}^1 = \emptyset$, and $G_p^1 = G_d$. Note that the *H* variables are used only to detect whether any *G* changed. Since G_p changed in value, we enter the loop again.

In the second iteration of the repeat-until loop, by substituting into the second of the above equations, we find that F_3^2 is:

$$\rho_{x/\$1,y/\$2,s_1/\$3,s_2/\$4}(\hat{\pi}_{x,y,s_1 s_2}(\rho_{\$1/x,\$2/z,\$3/s_1,\$4/s_3}\{g_1,g_2,g_3,g_4\}\bowtie \\ \rho_{\$1/z,\$2/y,\$3/s_3,\$4/s_2}\{g_1,g_2,g_3,g_4\}))$$

Here $\rho_{\$1/x,\$2/z,\$3/s_1,\$4/s_3}\{g_1,g_2,g_3,g_4\}$ is

$$\{(x=1, z=2, s_1 <_{19} s_3), (x=1, z=3, s_1 <_{44} s_3), (x=2, z=4, s_1 <_{29} s_3), \\ (x=3, z=4, s_1 <_{14} s_3)\}$$

and $\rho_{\$1/z,\$2/y,\$3/s_3,\$4/s_2}\{g_1,g_2,g_3,g_4\}$ is

$$\{(z=1, y=2, s_3 <_{19} s_2), (z=1, y=3, s_3 <_{44} s_2), (z=2, y=4, s_3 <_{29} s_2), \\ (z=3, y=4, s_3 <_{14} s_2)\}$$

The join of the above two will be:

$$\{(x=1, z=2, y=4, s_1 <_{19} s_3, s_3 <_{29} s_2), (x=1, z=3, y=4, s_1 <_{44} s_3, s_3 <_{14} s_2)\}$$

and after projection we get: $\{(x=1, y=4, s_1 <_{49} s_2), (x=1, y=4, s_1 <_{59} s_2)\}$

and after renaming we get: $\{(\$1=1, \$2=4, \$3 <_{49} \$4), (\$1=1, \$2=4, \$3 <_{59} \$4)\}$

Both of these gap-graphs will be added to the path relation. Similarly, F_2^2 will be:

$$\{(\$1=1, \$2=2, 0 <_{20} \$3), (\$1=1, \$2=3, 0 <_{45} \$3), (\$1=2, \$2=4, 0 <_{30} \$3), \\ (\$1=3, \$2=4, 0 <_{15} \$3)\}$$

We find that $G_p^2 = G_d \cup F_3^2$ and $G_{ns}^2 = F_2^2$. Since there are changes in the set of gap-graphs assigned to the IDB relations, we again enter the repeat-until loop.

In the third iteration of the repeat-until loop, similarly to the above, we find that $F_2^3 = F_2^2 \cup \{(\$1=1, \$2=4, 0 <_{50} \$3), (\$1=1, \$2=4, 0 <_{60} \$3)\}$, $F_3^3 = F_3^2$ and $F_4^3 = G_d$. We also find that $G_{ns}^3 = G_{ns}^2 \cup \{(\$1=1, \$2=4, 0 <_{50} \$3), (\$1=1, \$2=4, 0 <_{60} \$3)\}$ and $G_p^3 = G_p^2$. Since G_{ns} changed we enter the repeat-until loop again.

In the fourth iteration of the repeat-until loop, none of F s and G s will change. We exit the repeat-until loop and enter stratum 2.

In stratum 2 the only relation is *shortest*. To find the value of this relation, we have to enter again the repeat-until loop. Here in each iteration i we have:

$$F_1^i = \rho_{x/\$1,y/\$2,s/\$3}(\hat{\pi}_{x,y,s}(\rho_{\$1/x,\$2/y,\$3/0,\$4/s}G_p^{i-1}\bowtie(-(\rho_{\$1/x,\$2/y,\$3/s}G_{ns}^{i-1}))))$$

Let $S_1 = \rho_{\$1/x,\$2/y,\$3/0,\$4/s}G_p^4$. The gap-graphs in S_1 are:
 $(x=1, y=2, 0 <_{19} s)$
 $(x=1, y=3, 0 <_{44} s)$

$(x = 2, y = 4, 0 <_{29} s)$
 $(x = 3, y = 4, 0 <_{14} s)$
 $(x = 1, y = 4, 0 <_{49} s)$
 $(x = 1, y = 4, 0 <_{59} s)$

Let $S_2 = \rho_{\$1/x, \$2/y, \$3/s} G_{ns}^4$. The gap-graphs in S_2 are:

$(x = 1, y = 2, 0 <_{20} s)$
 $(x = 1, y = 3, 0 <_{45} s)$
 $(x = 2, y = 4, 0 <_{30} s)$
 $(x = 3, y = 4, 0 <_{15} s)$
 $(x = 1, y = 4, 0 <_{50} s)$
 $(x = 1, y = 4, 0 <_{60} s)$

We find the negation of S_2 using De Morgan's laws and simplifying:

$(s < 16)$
 $(x \neq 3, s < 21)$
 $(y \neq 4, s < 21)$
 $(x \neq 1, x \neq 3, s < 31)$
 $(x \neq 3, y \neq 2, s < 31)$
 $(x \neq 2, x \neq 3, y \neq 2, s < 46)$
 $(y \neq 2, y \neq 4, s < 46)$
 $(x \neq 2, x \neq 3, y \neq 2, y \neq 3, s < 51)$
 $(x \neq 1, x \neq 2, x \neq 3)$
 $(x \neq 1, y \neq 4)$
 $(y \neq 2, y \neq 3, y \neq 4)$

Each of the above constraint tuples can be rewritten into a set of gap-graphs by expanding the $\neq$ constraints into equivalent disjunctions, i.e. $x \neq 3$ into $(x > 3) \vee (x < 3)$. For simplicity we skip this step in the present example. It is already evident that the join of S_1 and the negation of S_2 will be:

$(x = 1, y = 2, s = 20)$
 $(x = 1, y = 3, s = 45)$
 $(x = 2, y = 4, s = 30)$
 $(x = 3, y = 4, s = 15)$
 $(x = 1, y = 4, s = 50)$

Note that we get a unique s for each pair of x and y . The s is the length of the shortest path between x and y as we expected. $\square$

5 The Complexity of Safe Stratified $Datalog^{\neg, <^z}$ Queries

Although safe stratified $Datalog^{\neg, <^z}$ queries can be evaluated in finite time, in this section we show that their evaluation may require a large data complexity. Since the language $Datalog^{<^z}$ is included in the language of safe stratified $Datalog^{\neg, <^z}$ it is worthwhile to recall the known results about this sublanguage.

(Note: Safe stratified $Datalog^{\neg, <^z}$ queries were not considered before in the literature.)

In [23] the data complexity of $Datalog^{<^z}$ queries is shown to be in PTIME if the size of each constant in the database is logarithmic in the size of the entire database and to be in DEXPTIME in general. In [11] the expression complexity in general is shown to be DEXPTIME-complete. In [24] the data complexity in general is also shown to be DEXPTIME-complete.

To proceed with the analysis of data complexity, we start with a definition of families of functions F_i of type $N \rightarrow N$. Let F_0 be the set of polynomial functions, and let $F_i = \{2^f : f \in F_{i-1}\}$ for $i > 0$. If $\mathcal{F}$ is a family of functions, let $\mathcal{F}$ -TIME denote the class of languages that can be accepted within some time $f \in \mathcal{F}$. Now we will show using a Turing machine reduction that evaluation of stratified $Datalog^{\neg, <^z}$ queries is $\mathcal{F}$ -TIME-hard.

Let d be a database instance and let $|d|$ denote its size in number of bits representation. Let $\mathcal{D}$ denote the set of possible database instances. We define a function f of type $N \times \mathcal{D} \rightarrow N$ as follows. Let $f(0, d) = 2^{|d|}$ and $f(i, d) = 2^{f(i-1, d)}$. (Here $f(i, d) \in \mathcal{F}_i$ -TIME.)

We start with a lemma that shows that the successor function on integers from 0 to $f(i, d)$ can be defined using a safe stratified $Datalog^{\neg, <^z}$ program with i strata.

Lemma 5.1 There is a safe stratified $Datalog^{\neg, <^z}$ program with a single negation that given as inputs a relation that enables counting from 1 to s , and the numbers s and 2^s defines both (1) a relation that enables counting from 1 to 2^s and (2) the numbers 2^s and 2^{2^s} .

Proof. Let us assume that the input relations are $next(0, 1), \dots, next(s-1, s)$ and $no_digits(s)$, $two_to_s(2^s)$. Using a safe stratified $Datalog^{\neg, <^z}$ program we will define two output relations, (1) the successor relation $succ(0, 1), \dots, succ(2^s-1, 2^s)$ and (2) the numbers $two_to_two_to_s(2^{2^s})$. In this abstract we show only (1) and omit (2).

To show (1): In the reduction it helps to think of each number being written in binary notation. Since the number 2^s has s binary digits, what we really need is given a counter on the digits define a counter from 1 to 2^s .

We start by representing the value of each digit using a constraint interval, where the gap-value is one less than the actual value. That is, for each $1 \leq i \leq s$, we want to represent the value of the i th digit from the right as: $digit(i, x_1, x_2) :- x_1 <_{2^{i-1}} x_2$. The following program P_1 will generate the desired constraint tuples.

$$\begin{aligned} digit(j, x_1, x_2) &:- next(i, j), digit(j, x_1, x_3), digit(j, x_3, x_2). \\ digit(1, x_1, x_2) &:- x_1 < x_2. \end{aligned}$$

Note that we can represent each number i by a pair of constraints: $-1 <_i x$ and $x <_{2^s-(i+1)} 2^s$. Since each number can be expressed as the sum of a subset of the values of the n digits, if we start out from the constraint $-1 < x$ and $x < 2^s$ and choose to increment for each $1 \leq i \leq s$ either the first or the second

gap-value by the value of the i th digit, then we will get a single integer between 0 and $2^s - 1$ as output. This gives an idea about how to “build up” any number that we need.

Using this idea, the following program defines all integers between 1 and 2^s . (The program is given here only as an illustration to the above idea, it is not used directly to express the successor function.)

$$\begin{aligned} \text{single_integer}(x) &:- \text{no_digits}(s), \text{range}(x, x, s). \\ \text{range}(x_3, x_2, j) &:- \text{next}(i, j), \text{range}(x_1, x_2, i), \text{digit}(j, x_1, x_3). \\ \text{range}(x_1, x_3, j) &:- \text{next}(i, j), \text{range}(x_1, x_2, i), \text{digit}(j, x_3, x_2). \\ \text{range}(x_1, x_2, 0) &:- -1 < x_1, x_2 < 2^s. \end{aligned}$$

Technical note: To avoid bad interactions we used a separate x_1 and x_2 in all the rules except the top-most. Intuitively, when computing with constraint tuples [23], in each recursive step, x_1 will be bounded by higher and higher constants from below and x_2 will be bounded by lower and lower constants from above. In the top rule the possible values of x_1 and x_2 will overlap exactly on one integer.

To express the succssor function, we build-up *pairs* of integers. Let x_1 and x_2 represent the first and y_1 and y_2 represent the second integer. Building up pairs at a time is necessary to make sure that when we add a digit to the x s we also add the same digit to the y s the right way.

$$\begin{aligned} \text{succ}(x, y) &:- \text{succ2}(x, x, y, y, s), \text{no_digits}(s). \\ \text{succ2}(x_3, x_2, y_3, y_2, j) &:- \text{succ2}(x_1, x_2, y_1, y_2, i), \text{next}(i, j), \text{digit}(j, x_1, x_3), \\ &\quad \text{digit}(j, y_1, y_3). \\ \text{succ2}(x_1, x_3, y_1, y_3, j) &:- \text{succ2}(x_1, x_2, y_1, y_2, i), \text{next}(i, j), \text{digit}(j, x_3, x_2), \\ &\quad \text{digit}(j, y_3, y_2). \\ \text{succ2}(x_1, x_3, y_3, y_2, 1) &:- \text{range}(x_1, x_2, y_1, y_2, 1), \text{digit}(1, x_3, x_2), \\ &\quad \text{digit}(1, y_1, y_3). \\ \text{succ2}(x_1, x_3, y_3, y_2, j) &:- \text{succ3}(x_1, x_2, y_1, y_2, i), \text{next}(i, j), \text{digit}(j, x_3, x_2), \\ &\quad \text{digit}(j, y_1, y_3). \\ \text{succ3}(x_3, x_2, y_1, y_3, j) &:- \text{succ3}(x_1, x_2, y_1, y_2, i), \text{next}(i, j), \text{digit}(j, x_1, x_3), \\ &\quad \text{digit}(j, y_3, y_2). \\ \text{succ3}(x_3, x_2, y_1, y_3, 1) &:- \text{range}(x_1, x_2, y_1, y_2, 1), \text{digit}(1, x_1, x_3), \\ &\quad \text{digit}(1, y_3, y_2). \\ \text{range}(x_1, x_2, y_1, y_2, 0) &:- -1 < x_1, x_2 < 2^s, -1 < y_1, y_2 < 2^s. \end{aligned}$$

Theorem 5.1 There is a fixed yes/no program Q in safe stratified $\text{Datalog}^{\neg, < \mathbf{z}}$ with i negations such that deciding whether $Q(d)$ is yes for variable database d is deterministic $\mathcal{F}_i$ – *TIME*-hard.

Proof. The base case, when $i = 0$, is just the case of $\text{Datalog}^{< \mathbf{z}}$ programs, which are known to have DEXPTIME-complete data complexity [24]. To prove the theorem for $i > 0$ we will show that we can simulate an $f(i, d)$ -time bounded deterministic Turing machine using a safe stratified $\text{Datalog}^{\neg, < \mathbf{z}}$ program with i negations. $\square$.

Theorem 5.1 shows that the data complexity of some stratified $Datalog^{\neg, <}$ programs can be high. This result of course means nothing about the data complexity of queries that an average user may wish to use. Therefore the high data complexity should not be considered a pessimistic result.

6 Conclusions

This paper considered only stratified $Datalog^{\neg, <}$ programs. It is still an open problem to find safe subsets of other similarly expressive constraint logic programming languages.

In addition it should be kept in mind that guaranteeing closed-form evaluation and bottom-up processing of queries are just two of the many important features that database systems today should have for enhanced usability and user satisfaction. For example, most current database systems also provide efficient indexing on facts, integrity constraints, built-in aggregate operators, menu-based user interfaces, concurrent access to data, security etc. Many of these problems and related issues have to be rethought in the context of constraint databases (see [22, 18, 13, 25] for some recent papers). The work in this paper is only a part of a bigger context of building a prototype constraint database system. We are in the process of implementing the algorithms presented in the paper and plan to demonstrate them at the conference.

References

1. H. Aït-Kaci, A. Podelski. Towards a Meaning of LIFE. *Journal of Logic Programming*, 16, 195–234, 1993.
2. K.R. Apt, H. Blair, A. Walker. Towards a Theory of Declarative Knowledge. In: *Foundations of Deductive Databases in Logic Programming*, Morgan-Kaufmann, 1988.
3. M. Baudinet, M. Niette, P. Wolper. On the Representation of Infinite Temporal Data and Queries. *Proc. 10th ACM PODS*, 280–290, 1991.
4. A. Brodsky, J. Jaffar, M. J. Maher. Toward Practical Constraint Databases, *Proc. VLDB*, 1993.
5. A.K. Chandra, D. Harel. Computable Queries for Relational Data Bases. *Journal of Computer and System Sciences*, 21:156–178, 1980.
6. A.K. Chandra, D. Harel. Structure and Complexity of Relational Queries. *Journal of Computer and System Sciences*, 25:99–128, 1982.
7. J. Chomicki, T. Imielinski. Finite Representation of Infinite Query Answers. *ACM Transactions of Database Systems*, 181–223, vol. 18, no. 2, 1993.
8. J. Chomicki, G. Kuper. Measuring Infinite Relations, *Proc. ILPS Workshop on Constraints and Databases*, 1994.
9. J. Cohen. Constraint Logic Programming Languages. *CACM*, 33 (7): 69–90, 1990.
10. A. Colmerauer. An Introduction to Prolog III, *CACM*, 28 (4): 412–418, 1990.
11. J. Cox, K. McAloon. Decision Procedures for Constraint Based Extensions of Datalog. In: *Constraint Logic Programming*, MIT Press, 1993.

12. M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. *Proc. Fifth Generation Computer Systems*, 1988.
13. S. Grumbach, J. Su. Finitely Representable Databases. *Proc. 13th ACM PODS*, 289–300, 1994.
14. J. Jaffar, J.L. Lassez. Constraint Logic Programming. *Proc. 14th ACM POPL*, 111–119, 1987.
15. J. Jaffar, M.J. Maher. Constraint Logic Programming: A Survey. *J. Logic Programming*, 19 & 20, 503–581, 1994.
16. J. Jaffar, S. Michaylov, P.J. Stuckey, R.H. Yap. The CLP(R) Language and System. *ACM Transactions on Programming Languages and Systems*, 14:3, 339–395, 1992.
17. F. Kabanza, J-M. Stevenne, P. Wolper. Handling Infinite Temporal Data. *Proc. 9th ACM PODS*, 392–403, 1990.
18. P.C. Kanellakis, D.Q. Goldin. Constraint Programming and Database Query Languages. *Proc. 2nd TACS*, 1994.
19. P. C. Kanellakis, G. M. Kuper, P. Z. Revesz. Constraint Query Languages. *Proc. 9th ACM PODS*, 299–313, 1990. Final version to appear in *Journal of Computer and System Sciences*.
20. P.C. Kanellakis, S. Ramaswamy, D.E. Vengroff, J.S. Vitter. Indexing for Data Models with Constraints and Classes *Proc. 12th ACM PODS*, 233–243, 1993.
21. M. Koubarakis. Complexity Results for First-Order Theories of Temporal Constraints. *Proc. Int. Conf. on Knowledge Representation and Reasoning*, 1994.
22. *Proc. Post-ILPS Workshop on Constraints and Databases*, P.Z. Revesz, D. Srivastava, P. Stuckey, S. Sudarshan (eds.). Tech. Report, Univ. of Nebraska, CSE-94-025, 1994.
23. P. Z. Revesz. A Closed Form Evaluation for Datalog Queries with Integer (Gap)-Order Constraints, *Theoretical Computer Science*, vol. 116, no. 1, 117–149, 1993.
24. P. Z. Revesz. Datalog Queries of Set Constraint Databases. *Proc. 5th International Conference on Database Theory*, 423–438, 1995.
25. D. Srivastava, R. Ramakrishnan, P.Z. Revesz. Constraint Objects. *Proc. 2nd Workshop on Principles and Practice of Constraint Programming*, 274–284, 1994.
26. D. Toman, J. Chomicki, D.S. Rogers. Datalog with Integer Periodicity Constraints. *Proc. International Logic Programming Symposium*, 189–203, 1994.
27. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.
28. J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, Vols 1&2, Computer Science Press, 1989.
29. M. Vardi. The Complexity of Relational Query Languages. *Proc. 14th ACM Symposium on the Theory of Computing*, 137–145, 1982.