[1] **1** [2]**1** **1.55em** **2**
1[2] **1.55em** **2**

Datalog Programs with Difference Constraints

Peter Z. Revesz
University of Nebraska–Lincoln
E-mail:revesz@cse.unl.edu.

Abstract: Datalog programs with difference constraints, (i.e., pure Horn clause logic programs without any function symbols and constraints except for constraints of the form $x - y \geq c$ over the integers) can express the entire r.e. class, hence their safety (i.e, termination on every input database) is undecidable in general. We use two new approaches to deal with this. First, we certify that specific programs are safe for any valid database input. For this end, we introduce *typed* Datalog programs. The type declarations and the program structure can often lead to *typed-safety*, a decidable property that implies safety. Algorithms are presented for checking whether a typed program is typed-safe and for evaluating the least fixpoint of typed-safe queries.
Second, we provide approximate evaluations on the least fixpoint of unsafe queries.

1 Introduction

The halting problem is a fundamental problem in computer science that has plagued general purpose programming languages from the very beginning of programming. Database programming languages were supposed to have avoided this problem by being restricted languages in which all queries terminate. As database programming languages have been extended with various constraints since the initial work on constraint query languages by Kanellakis, Kuper and Revesz [6] the halting problem has resurfaced. There are four options for dealing with safety.
I. **Ignore the issue.** In constraint databases the long-term goal is to provide something for naive users who have no idea of the halting problem. They may be very frustrated if they encounter a non-terminating query without any warning. Hence this is not a good option.
II. **Insist on safe query languages** (i.e., in which every program is terminating on every input database). This has been the approach used up to now in constraint databases. Figure 1 summarizes some results in this line of research. In the figure D means any discrete domain. In general [13] presents a sufficient restriction on the type of constraints allowed to obtain safe query languages considering standard fixpoint semantics, but see [4] and [10] for some other results on safe recursive queries going beyond standard semantics.
III. **Certify safe programs.** In this approach a database user is allowed to use a very expressive database programming language. If a program written by a user can be shown to terminate on every valid input database, then it is certified to be safe. A user may continue to use an unsafe program at his/her own risk after receiving a warning from the system. We take this approach in Section 4. The main idea is to derive an easily testable sufficient termination condition that is different from the termination conditions given for example in [2] and [3] for the top-down evaluation of constraint logic programs.
IV. **Approximate the output.** In this approach, the output of the unsafe program is approximated. We take this approach in Section 5.

Domain	Constraint	Datalog	See
set of D	$=_{Bool}$	any	[6]
set of D	$\subseteq, \neq_{monBool}$	any	[9]
integer	gap-order	any	[7]
integer	modulus	any	[12]
rational	linear	positive,negative	[11]
real	polynomial	non-recursive	[6]

Figure1: Safe Query Languages

We consider difference constraints of the form $x - y \geq c$ where c can be any integer. Datalog$^{\text{diff}}$ programs can express the entire r.e. class. This is desirable from the point of expressive power, but it also implies that safety is undecidable for Datalog$^{\text{diff}}$ programs. Therefore, we introduce a weaker form of safety, called *typed-safety* that applies to *typed* Datalog programs. Any typed-safe program is safe, but not all safe programs are typed-safe. The notion of typed-safety is decidable for typed Datalog$^{\text{diff}}$ programs.

Each typed Datalog$^{\text{diff}}$ program is a Datalog$^{\text{diff}}$ program together with a type declaration for each input relation. This type declaration indicates something about the syntax of the constraints that are allowed in a constraint relation. Our motivation for typing is the following. A program that does not terminate on some type A of input databases may terminate on some other type B of input databases. Similarly, a program may be typed-safe on A but not typed-safe on B.

This paper is organized as follows. In Section 2 we present some basic definitions. In Section 3 we define an algebraic evaluation of the least fixpoint of

Datalog$^{\text{diff}}$ programs and prove a sufficient condition for termination. In Section 4 we define typed-safe Datalog$^{\text{diff}}$ programs and show that they satisfy the sufficient condition for termination. In Section 5 we give an approximate evaluation method for the least fixpoint of any (safe or unsafe) Datalog$^{\text{diff}}$ program. The approximation method gives a lower and an upper bound for the least fixpoint.

2 Basic Concepts

2.1 Constraints

A *difference constraint* has the form

$$x - y \geq c$$

where x and y are variables or the constant 0 and c is a constant. We call c the *bound* or right hand side and $x-y$ the left hand side in each difference constraint. We assume that the domain of the variables and constants is the set of integer numbers, denoted by $\mathbf{Z}$.

Note that lower bound constraints $x \geq c$, upper bound constraints $x \leq c$, equality constraints $x = y$ and $x = c$, and addition constraints $x = y + c$ can be all expressed by (conjunctions of) difference constraints. Hence we do not deal with these separately in this paper.

2.2 Datalog$^{\text{diff}}$

Constraint Tuples: Each input database is a set of constraint tuples that have the form,

$$R_0(x_1, \ldots, x_k) \text{ :— } \psi.$$

where ψ is a conjunction of difference constraints on $x_1, \ldots, x_k$ which are not necessarily distinct variables or constants.

Rules: Each Datalog program is a set of rules that have the form,

$$\begin{array}{lll} R_0(x_1, \ldots, x_k) & \text{:—} & R_1(x_{1,1}, \ldots, x_{1,k_1}), \\ & & \ldots, \\ & & R_n(x_{n,1}, \ldots, x_{n,k_n}), \\ & & \psi. \end{array}$$

where $R_0, \ldots, R_n$ are not necessary distinct relation symbols and the xs are not necessarily distinct variables or constants and ψ is a conjunction of difference constraints. We call the left hand side of :— the *head* and the right hand side of :— the *body* of the rule. Several constraint tuples or several rules can have the same left-hand relation name. In the constraint tuples all variables in the body also appear in the head. In the rules some variables in the body may not appear in the head.

Query: Each Datalog query consists of a Datalog program and an input database.

Semantics: Let Q be any Datalog query with constraints. We call an *interpretation* of Q any assignment I of a finite or infinite number of tuples over $\mathbf{Z}^{\alpha(R_i)}$ to each R_i that occurs in Q, where $\alpha(R_i)$ is the arity of relation R_i.

The *immediate consequence operator* of a Datalog query Q, denoted T_Q, is a mapping from interpretations to interpretations as follows. For each interpretation I: $R_0(a_1, \ldots, a_k) \in T_Q(I)$ iff there is an instantiation ν of all variables by constants from $\mathbf{Z}$, including variables $x_1, \ldots, x_k$ by constants $a_1, \ldots, a_k$, in either a constraint tuple of the form above such that $\nu(\psi)$ is true, or a rule of the form above such that $R_i(\nu(x_{i,1}, \ldots, x_{i,k_i})) \in I$ for each $1 \leq i \leq n$ and $\nu(\psi)$ is true.

Let $T_Q^0(I) = T_Q(I)$. Also let $T_Q^{i+1}(I) = T_Q^i(I) \cup T_Q(T_Q^i(I))$. An interpretation I is called a *least fixpoint* of a query Q iff $I = \bigcup_i T_Q^i(I)$.

Each difference constraint relation R represents a possibly infinite set of tuples of constants that satisfy R. This set we denote by $Mod(R)$.

3 Evaluation of Datalog$^{\text{diff}}$ Programs

In this section we show a quantifier-elimination-based and an algebraic evaluation of Datalog$^{\text{diff}}$ queries. We also give a sufficient termination condition. Section 4 will investigate when this termination condition can be guaranteed. We describe the steps of the evaluation method as follows.

Constraint Rule Application: Let us assume that we have a rule of the general form in Section 2 and we also have input or derived constraint tuples for each $1 \leq i \leq n$ of the form:

$$R_i(x_{i,1}, \ldots, x_{i,k_i}) \text{ :— } \psi_i(x_{i,1}, \ldots, x_{i,k_i}).$$

where formula ψ_i is a conjunction of constraints. A *constraint rule application* of this rule given these constraint tuples as input produces the following derived constraint tuple:

$$R_0(x_1, \ldots, x_k) \text{ :— } \phi(x_1, \ldots, x_k).$$

where ϕ is a quantifier-free formula that is equivalent to

$$\exists * \psi_1(x_{1,1}, \ldots, x_{1,k_1}), \ldots, \psi_n(x_{n,1}, \ldots, x_{n,k_n}), \psi.$$

where $*$ is the list of the variables in the body of the rule which do not occur in the head of the rule.

Remark: For Datalog^{diff} queries the immediate consequence operator may not be computationally effective because there may be an infinite possible number of substitutions for the variables in a rule. In contrast, constraint rule applications which can compute non-ground output tuples are always effectively computable.

The *bottom-up constraint fixpoint evaluation* of Datalog queries starts from the input constraint tuples and rules and repeatedly applies one of the rules until no

new constraint tuples can be derived and added to the database. We call the set of input and derived constraint tuples the constraint least fixpoint of the query.

Note that a constraint tuple is equivalent to a possibly infinite number of regular tuples of constants from the domain. Hence a finite number of constraint tuples could represent an infinite least fixpoint. In fact, similar to other cases of constraint Datalog programs [6], it can be shown that:

Proposition 3.1 *For any Datalog$^{\text{diff}}$ query the bottom-up constraint least fixpoint is equivalent to the least fixpoint.* □

The next lemma shows that existentially quantified variable elimination can be done.

Lemma 3.1 *Let S be any conjunction of difference constraints over the variables $x, y_1, \ldots, y_n$. Then we can rewrite $\exists x S$ into a logically equivalent S' of difference constraints over $y_1, \ldots, y_n$.*

Proof: S' will be the conjunction of all the difference constraints in S that do not contain the variable x and all the difference constraints that can be derived from any pair of difference constraints in S using the implication below for any y_i and y_j where $y_0 = 0$ for $0 \leq i, j \leq n$.

$y_i - x \geq c_1$ and $x - y_j \geq c_2$ imply $y_i - y_j \geq c_1 + c_2$

Given any two difference constraints with opposite signs for x, their sum is returned by the implication. It is easy to see that if S consisted of difference constraints, then S' will contain only difference constraints. The only case that merits special mention is when y_i and y_j are the same variables. In that case we will obtain 0 on the left hand side. If the bound on the right hand side is less than or equal to 0, then this constraint can be dropped, else we can return a flag that S' is unsatisfiable.

For any instantiation, if two difference constraints are both true, then their sum also must be a true difference constraint. Hence if S is true, then S' must be also true for any instantiation of the variables $x, y_1, \ldots, y_n$.

For the other direction, suppose that S' is true for some instantiation of the variables $y_1, \ldots, y_n$. Then make the same instantiation into S. After the instantiation, x will be the only remaining variable in S. Wherever x occurs positively, the constraint implies a lower bound for x, and wherever x occurs negatively the constraint implies an upper bound for x.

Suppose that the largest lower bound l is implied by some constraint f and the smallest upper bound u is implied by some constraint g in S. Since the sum of f and g under the current instantiation is equivalent to $l \leq u$ and is in S', which is true, we can find a value between l and u inclusively for x that will make S also true. □

Although it would be possible to perform constraint rule applications using Lemma 3.1, we would like to give constraint rule applications more structure by considering an algebraic evaluation of the rule bodies. That would lead to an algebraic rule application that is equivalent to a set of constraint rule applications performed in parallel.

Let S be any conjunction of difference constraints. We simplify S so that it contains at most one difference constraint with each different left hand side. If S has several difference constraints with the same left hand side all but the one with the highest bound is superfluous and is deleted. We call the remaining set of constraints the *normal form* of S.

Example 3.1 *Let S be $x - y \geq 5, x - y \geq 8, y - z \geq 2, z - y \geq 4$, then we have three different left hand sides, namely $x - y$ and $y - z$ and $z - y$. From the two constraints with the same left hand side $x - y$, we only keep the stronger one, i.e. $x - y \geq 8$. Hence the normal form of S will be $x - y \geq 8, y - z \geq 2, z - y \geq 4$.*

Normal form conjunction of difference constraints can be represented by a labeled directed graph, called a *difference graph* or *d-graph*. The d-graph contains a vertex for each variable and the constant 0 and a labeled directed edge from y to x with label c for each constraint of the form $x - y \geq c$.

Each tuple of a relation R can be represented by a d-graph, and each relation R can be represented by a set of d-graphs over the same set of vertices. Next we define the following algebraic operations on sets of d-graphs.

1. **Rename:** If relation R has d-graphs G and ρ is a renaming of some of the variables by other variables, then the d-graphs of $\rho(R)$ are $\rho(G)$.

2. **Select:** If relation R has d-graphs G and σ is a selection of the form $x - y \geq c$, then the d-graphs of $\sigma(R)$ are

   $$\{g \cup \{(y, x)^c\} \mid g \in G\}$$

   where $(y, x)^c$ is an edge from y to x with label c.

   In each g if there is an edge $(y, x)^d$ with $d \leq c$, then delete that edge from g.

   Test for satisfiability of each d-graph. Delete the unsatisfiable d-graphs.

3. **Project:** If relation R has d-graphs G and R' is a projection of R onto all variables except x_i, then the d-graphs of R' are

   $$\{elim(x_i, g) \mid g \in G\}$$

   where $elim(x_i, g)$ is the d-graph that is obtained from g in two steps. In the first step, for each pair

of x_j and x_k add edge $(x_j, x_k)^{c_1+c_2}$ to g if $(x_j, x_i)^{c_1}$ and $(x_i, x_k)^{c_2}$ are edges in g. In the second step, delete x_i and all edges adjacent to it. If there are several edges between two vertices, delete all of them except the one with the maximum label.

Test for satisfiability of each d-graph. Delete the unsatisfiable d-graphs.

4. **Join:** If R_1 and R_2 are relations with d-graphs G_1 and G_2, then the join of them is a relation that has d-graphs

$$\{g(V_1 \cup V_2, E_1 \cup E_2) \quad | \quad g_1(V_1, E_1) \in G_1, \; g_2(V_2, E_2) \in G_2\}$$

The label of an edge is the maximum of the labels in either E_1 or E_2.

Test for satisfiability of each d-graph. Delete the unsatisfiable d-graphs.

3.1 Sufficient Termination Condition

We next present a sufficient condition for termination.

Theorem 3.1 *For any l, if the bottom-up constraint fixpoint evaluation of a Datalog$^{\text{diff}}$ program does not create any constraint tuple containing a bound less than l, then there is a modified evaluation that returns the constraint fixpoint in finite time.*

Proof: Let us consider any output relation $R(x_1, \ldots, x_{n-1})$. Clearly, it does not affect the correctness of the evaluation if we normalize every derived constraint tuple before adding it to R.

In every normalized tuple (equivalently d-graph) for R there are clearly only n^2 different left hand sides (directed edges).

Let us fix any ordering of the n^2 possible left hand sides. Using this fixed ordering, we can represent any normal form constraint tuple S as a n^2-dimensional point in which the ith coordinate value will be $(c+l+1)$ if S contains a difference constraint with the ith left hand side and bound c, and 0 if S does not contain any difference constraint with the ith left hand side.

Using this representation, the condition that the evaluation does not create a bound less than l can be translated as saying that the evaluation only creates points that are represented with only non-negative integer coordinates. Therefore the sequence of derived constraint tuples for R can be represented using a point sequence:

$p_1, p_2, \ldots$

We say that a point *dominates* another point if it has the same dimension and all of its coordinate values are $\geq$ the corresponding coordinate values in the other point.

It is easy to see that if point p_i dominates point p_j, then p_i and p_j represent conjunctions S_i and S_j of difference constraints such that the set of solutions of S_i is included in the set of solutions of S_j. This shows that the fixpoint evaluation could be modified to add only points that do not dominate any earlier point in the sequence. By the geometric Lemma in [7], in any fixed dimension any sequence of distinct points with non-negative integer coordinates must be finite, if no point dominates any earlier point in the sequence. □

4 Typed Datalog$^{\text{diff}}$ Programs

Tuple types are graphs that are similar to d-graphs except that the labels on the edge are abstracted and represented by colors. That is, a *tuple type* is a directed graph whose vertices are labeled by variables and the constant 0 and whose edges are colored blue, purple or red.

In a type, a directed edge from y to x indicates that there *may be* a constraint of the form $x - y \geq c$. If the edge is red then the bound must be non-negative. If it is blue, then the bound could be any integer.

More precisely, we say that a constraint tuple t *has type* g with respect to a negative integer l, denoted by $t :_l g$, if whenever there is a constraint of the form $x - y \geq c$ in t, then there is a directed edge from y to x in g. Further, if the edge is red, then $c \geq 0$, and if it is blue, then $c \geq l$.

Example 4.1 *Let t be the constraint tuple $R(x, y, z, u) :\!-\; x - y \geq -4, \;\; y - z \geq 1, \;\; z - u \geq -2$.*

Also, let $g(V, E)$ be a type with vertices $V = \{0, x, y, z, u\}$ and edges $E = \{(y, x)^p, (z, y)^r, (u, z)^b\}$ where superscripts b, p, r indicate that an edge is blue, purple or red, respectively.

Then $t :_{(-3)} g$ is true. □

A *relation type* is a set of tuple types.

We say that a constraint relation R *has type* G with respect to an integer l, denoted by $R :_l G$, if for each constraint tuple t of R there is a $g \in G$ such that $t :_l g$.

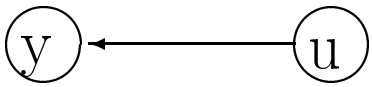
A *typed program* is a Datalog$^{\text{diff}}$ program with a type declaration for each input constraint relation.

Example 4.2 *The following typed Datalog$^{\text{diff}}$ program has input relations P and Q and output relation R. The rules are the following.*

$R(x, z, v) \quad :\!-\quad z - x \geq 50, \;\; v - z \geq -90.$

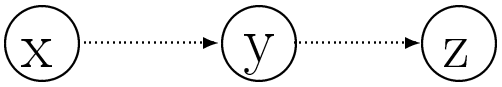
$R(x, z, v) \quad :\!-\quad P(u, y), \;\; Q(x, y, z), \;\; R(z, u, v).$

Suppose that the type declaration with respect to $l = -100$ of relation P is G_P which contains a single graph g_1:



*where (***dashed line = red***) and (***solid line = blue***).*

Similarly, the type declaration of Q is G_Q contains only the graph g_2:



Note: The choice of these type declarations is arbitrary. We could have also declared other types. □

Each typed Datalog$^{\text{diff}}$ program will accept only inputs that have the declared types. Therefore, each execution starts with checking whether the input database has the right type. Hence, the following is important:

Lemma 4.1 *Let $R(x_1, \ldots, x_k)$ be any fixed relation scheme. Then for any instance r with difference constraints and type declaration G of R and any integer l, it can be checked in linear time in the size of r and G whether $r :_l G$.*

Proof: Each tuple type in G can be simplified by deleting vertices other than $x_1, \ldots, x_k$ and edges adjacent to them. If the vertices $x_1, \ldots, x_k$ do not occur in a tuple type, then they can be added to it as isolated vertices.

After the simplification we will have exactly $k+1$ vertices and we can have $(k+1)^2$ edges which are either blue, red, purple or absent. Therefore, there can be only $O(4^{(k+1)^2})$ different tuple types in the worst case, that is, some constant c. For each of the tuples in r we have to check whether it has one of the types remaining in G. Therefore, if r contains n tuples, then there are $O(n)$ number of checks whether a tuple has a tuple type.

For checking whether a tuple t has a tuple type g, we can do the following. We read the constraints in t in sequence. For each constraint of the form $x - y \geq c$ we check that there is an edge in g from y to x. If c is negative we also check that the color of the edge is blue or purple. If c is less than l, we check that the color of the edge is purple. Clearly, this yields a linear time algorithm for type checking. □

We introduce a precedence relationship among the three colors. We say that red is greater than blue and blue is greater than purple.

When a query is evaluated, each output relation will have a type that is dependent on the program and the type of the input relations. For typed rename-select-project-join relational algebra expressions we define inductively on the structure of the expressions the output relation type with respect to a fixed l throughout as follows.

1. **Rename:** If relation $R :_l G$ and ρ is a renaming of some of the variables by other variables, then $\rho(R) :_l \rho(G)$.

2. **Select:** If relation $R :_l G$ and σ is a selection of the form $x - y \geq c$, then the type of $\sigma(R)$ with respect to l is found as follows.

   If $c \geq 0$, then add $(y, x)^r$, else if $c \geq l$, then add $(y, x)^b$, else add $(y, x)^p$. If edge (y, x) has after the addition several colors, then preserve only the maximum color.

3. **Project:** If relation $R :_l G$ and R' is a projection of R onto all variables except x_i, then the type of R' with respect to l is

   $$\{elim(x_i, g) \mid g \in G\}$$

   where $elim(x_i, g)$ is the graph that is obtained from g in two steps. In the first step, for each pair of x_j and x_k add edge (x_j, x_k) to g if (x_j, x_i) and (x_i, x_k) are edges in g. If both of the latter edges are red then color the new edge red, else if one of them is red and the other is blue, color it blue, else color it purple. In the second step, delete x_i and all edges adjacent to it. If there are several edges between two vertices, delete all of them except the one with the maximum color. For example, a red edge implies that $b \geq 0$ and a blue edge implies that $b \geq l$. The conjunction of these implies that $b \geq 0$, hence we keep the red edge, which indeed has a higher precedence than the blue edge.

4. **Join:** If $R_1 :_l G_1$ and $R_2 :_l G_2$, then the join of them is a relation that has type with respect to l

   $$\{g(V_1 \cup V_2, E_1 \cup E_2) \mid g_1(V_1, E_1) \in G_1, \; g_2(V_2, E_2) \in G_2\}$$

   The color of an edge is red if it is red in either E_1 or E_2, else it is blue if it is blue in either, else it is purple.

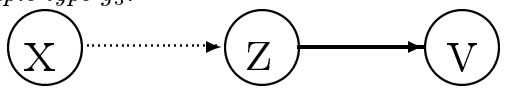
For Datalog$^{\text{diff}}$ programs we define the types of the output relations by algorithm *Find-Datalog-Types* shown below. (We assume that the Datalog$^{\text{diff}}$ program is in a rectified form, that is, each defined relation p appears in the head of rules with the same list of argument variables. Rectification is a minor restriction because any Datalog program can be written in an equivalent rectified form [1].)

Algorithm Find-Datalog-Types
INPUT: A typed Datalog$^{\text{diff}}$ program.
OUTPUT: The types of output relations.

FOR each output relation p_m **DO**
    assign to p_m the type $\emptyset$
END-FOR
WHILE any changes in types **DO**
    FOR each rule r with head p_m **DO**
        Find type G_r of the rule body using
        $\rho, \sigma, \pi, \bowtie$ definitions above.
        Union G_r to the type of p_m.
    END-FOR
END-WHILE
RETURN(types of output relations)

Example 4.3 *We illustrate the above definitions by finding with respect to $l = -100$ the type G_R of the output relation R of the Datalog$^{\text{diff}}$ program in Example 4.2.*
Initialization: $G_R = \emptyset$.

First Iteration: *The body of the first rule is equivalent to two selection operations on the relation that contains all tuples over x, z, v. Hence we add to G_R the following tuple type g_3:*



The body of the second rule is equivalent to the relational algebra expression:

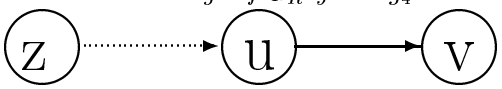
$$\pi_{x,z,v}(\pi_{x,z,v,y}(G_P \bowtie G_Q \bowtie (\rho_{x/z}(\rho_{z/u}(G_R)))))$$

where the double projection is needed because our definition of projection allows projection of one variable at a time. Since initially G_R is empty, the evaluation of the rule body yields only an empty relation.

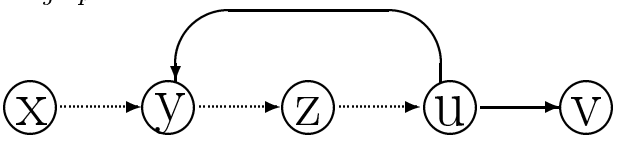
Second Iteration: *The first rule does not add any new tuple types.*

The rule body of the second rule is now evaluated as follows.

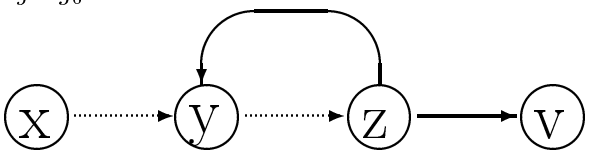
The two renamings of G_R yields g_4:



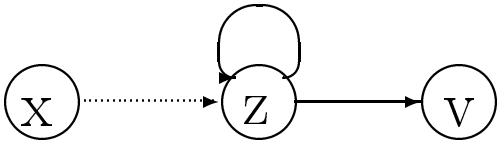
Joining G_P, G_Q and the renamed G_R, that is g_1, g_2 and g_4 yields g_5 by taking the union of the edges in the three graphs:



The projection $\pi_{x,z,v,y}$ is done by eliminating u, and we get g_6:



The projection $\pi_{x,z,v}$ is done by eliminating y, and we get g_7:



Hence g_7 is added to G_R.

Third Iteration: *The first rule does not add any new tuple types.*

The evaluation of the second rule will be similar as in the second iteration, except that instead of g_4 and g_5 we get graphs that also contain a self-loop from u to u. Therefore, nothing new is added to G_R in the third iteration. Hence the algorithm exits the while loop and returns the type $G_R = \{g_3, g_7\}$. □

The types are used to define *typed-safe* programs as follows.

Definition 4.1 *A typed Datalog$^{\text{diff}}$ program is* typed-safe *if the types of its output relations do not contain any purple edges.*

The following lemma proves that the calculation of the output relation types and testing for typed-safety for Datalog programs can be done efficiently.

Lemma 4.2 *For any fixed k, it can be checked in PTIME in the size of a Datalog$^{\text{diff}}$ program with maximum k-arity relations whether it is typed-safe.* □

Example 4.4 *The program of Example 4.2 is typed-safe because as we saw in Example 4.3 none of the types in G_R contains a purple edge. (Note that this program is not a Datalog with gap-order program[7], hence it could not be proved safe by previous results.)* □

Remark: Our main goal is to prove that given Datalog^{diff} programs are safe with respect to *all databases* of a certain type. However, it is also possible to check whether a given Datalog^{diff} program is safe on a new database input without the user specifying the input database types. To do this, we simply find for each input relation R a type (i.e., for each d-graph in R a tuple type and then taking union) and also take l to be the minimum of zero and the smallest constant that occurs in any difference constraint in the query, and then proceeding as above.

4.1 Termination of Typed-Safe Programs

In the following, we distinguish between the algebraic operators on types by writing them as $\dot{\rho}, \dot{\sigma}, \dot{\pi}, \dot{\bowtie}$ and the algebraic operators on sets of d-graphs by writing them as $\hat{\rho}, \hat{\sigma}, \hat{\pi}, \hat{\bowtie}$. We will drop the accent marks when it is obvious from the context which set of operators are applied. We can show that the operators on sets of d-graphs defined are semantically correct in the following sense.

Lemma 4.3 *The following are true for any difference constraint relations R, R_1, R_2, rename condition B, select condition C, and list of variables X, where $\rho, \sigma, \pi, \bowtie$ are the standard rename, select, project and join operators over unrestricted (finite or infinite) relations.*
(1) $Mod(\hat{\rho}_B(R)) \equiv \rho_B(Mod(R))$
(2) $Mod(\hat{\sigma}_C(R)) \equiv \sigma_C(Mod(R))$
(3) $Mod(\hat{\pi}_X(R)) \equiv \pi_X(Mod(R))$
(4) $Mod(R_1 \hat{\bowtie} R_2) \equiv Mod(R_1) \bowtie Mod(R_2)$ □

The next lemma shows that the operations defined above yield always output relations that have the defined types.

Lemma 4.4 (Correctness of Rename-Select-Project-Join Type Definitions) *The following are true for any constraint tuples t, t_1, t_2, types g, g_1, g_2, integer l and rename ρ, select σ, project π and join $\bowtie$ operators.*
(1) $t :_l g$ then $\hat{\rho}(t) :_l \dot{\rho}(g)$.
(2) $t :_l g$ then $\hat{\sigma}(t) :_l \dot{\sigma}(g)$.
(3) $t :_l g$ then $\hat{\pi}(t) :_l \dot{\pi}(g)$.
(4) $t_1 :_l g_1$ and $t_2 :_l g_2$ then $t_1 \bowtie t_2 :_l g_1 \bowtie g_2$. □

Based on Lemma 4.4 we can say the following about Datalog$^{\text{diff}}$ programs.

Lemma 4.5 (Correctness of Datalog Type Definitions) *The following is true for any typed Datalog$^{\text{diff}}$ program with input relations $R_1, \ldots, R_m$ with types $G_1, \ldots, G_m$ and output relation R_0 with type G_0 (as computed by algorithm Find-Datalog-Types) and integer l after any number of rule applications.*
If $R_i :_l G_i$ *for* $1 \leq i \leq m$ **then** $R_0 :_l G_0$. □

Now we can show termination for typed-safe Datalog$^{\text{diff}}$ programs.

Theorem 4.1 *The least fixpoint of any typed-safe Datalog$^{\text{diff}}$ query is evaluable in difference constraint form when the domain is the integers.*

Proof: By Theorem 3.1 it is enough to show that the evaluation of typed-safe Datalog$^{\text{diff}}$ programs does not create constraint tuples with a bound less than l for some l.

We take l to be the smallest bound in either the program or the input database. Then by Lemma 4.5 all the output relations satisfy their types. Further, by the definition of typed-safe programs the types do not contain purple edges. Hence the output relations must contain only constraints that have a bound $\geq l$. □

5 Approximate Evaluation for unsafe Datalog$^{\text{diff}}$

While typing can greatly increase the set of Datalog$^{\text{diff}}$ programs that can be used, there are many programs that are still unsafe. For these programs, we can do an approximate evaluation.

First, we note that what unsafe typed programs have in common is that they contain types with purple edges. Purple edges are created when two blue edges are transitively composed into a single edge. Since the blue edges indicate the possibility of negative bounds in the corresponding constraint, the purple edges indicate the possibility of creation of a negative bound that is lower than any bound initially occurring in the program or any input relation.

Once a purple edge is created it may be further composed by other purple or blue edges leading to the creation of even smaller bounds. This leads to the idea of placing a limit l on the allowed smallest bound. To avoid smaller bounds than l, we may do two different modifications to the evaluation method.

Modification 1: We modify the evaluation method by adding to any output database relation a new constraint tuple only after changing the value of any bound c to be $\max(c, l)$.

Modification 2: We modify the evaluation method by adding to any output database relation a new constraint tuple only after deleting from it any difference constraint that has a bound less than l.

Let $lfp(\Pi(D), l\uparrow)$ and $lfp(\Pi(D), l\downarrow)$ denote the output of the first and the second modified evaluation algorithms, respectively. We can show the following.

Theorem 5.1 *For any Datalog$^{\text{diff}}$ program Π, input database D and constant l, the following is true:*

$$lfp(\Pi(D), l\uparrow) \subseteq lfp(\Pi(D)) \subseteq lfp(\Pi(D), l\downarrow)$$

Further, $lfp(\Pi(D), l\uparrow)$ and $lfp(\Pi(D), l\downarrow)$ can be evaluated in finite time.

Proof: In the case of the first modification, the modified tuple implies the original one. Therefore, after any number of rule applications the output relations obtained by the modified algorithm also imply the output relations obtained by the original algorithm.

In the case of the second modification, the original tuple implies the modified one. Therefore, after any number of rule applications the output relations obtained by the original algorithm also imply the output relations obtained by the modified algorithm.

In both cases by Theorem 3.1 there is a modified evaluation that terminates. (In both cases, we need also to add dominance checking as within the proof of Theorem 3.1.) □

We can use the above to get better and better approximations using smaller and smaller values as bounds. In particular,

Theorem 5.2 *For any Datalog$^{\text{diff}}$ program Π, input database D and constants l_1 and l_2 such that $l_1 \leq l_2 \leq 0$, the following hold.*

$$lfp(\Pi(D), l_2\uparrow) \subseteq lfp(\Pi(D), l_1\uparrow)$$

$$lfp(\Pi(D), l_1\downarrow) \subseteq lfp(\Pi(D), l_2\downarrow)$$

□

6 Conclusions

We studied two new approaches to the termination problem in constraint databases. These approaches greatly extend the set of queries that users may safely use. An open problem is to design and implement constraint database systems with menu interfaces on top of the logical languages studied here, similar to the development of relational databases which implement menu interfaces on top of SQL. The constraint query languages then could be provided for naive users as was the original aim in [6]. There are several system implementations that make good progress towards that goal [10].

Acknowledgement: This work was supported in part by the USA NSF grants IRI-9625055 and IRI-9632871 and a Gallup Research Professorship.

References

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

[2] L. Colussi, E. Marchiori, M. Marchiori. On Termination of Constraint Logic Programs. *Proc. First International Conference on Principles and Practice of Constraint Programming*, Springer-Verlag LNCS 976, 431–448, 1995.

[3] Giacobazzi, R., Debray, S.-K., Levi, G. Generalized Semantics and Abstract Interpretation for Constraint Logic Programs. *Journal of Logic Programming*, vol. 25, no. 3, pp. 191–247, 1995.

[4] S. Grumbach, G. Kuper. Tractable Recursion over Geometric Data, *Proc. International Conference on Principles and Practice of Constraint Programming*, Springer-Verlag LNCS 1330, 450-462, 1997.

[5] J. Jaffar, J.L. Lassez. Constraint Logic Programming. *Proc. 14th ACM Symposium on Principles of Programming Languages*, 111–119, 1987.

[6] P. C. Kanellakis, G. M. Kuper, P. Z. Revesz. Constraint Query Languages. *Journal of Computer and System Sciences*, vol. 51, no. 1, pp. 26-52, 1995.

[7] 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.

[8] P. Z. Revesz. Safe Query Languages for Constraint Databases, *ACM Transactions on Database Systems*, vol. 23, no. 1, pp. 58-99, 1998.

[9] P. Z. Revesz. The Evaluation and the Computational Complexity of Datalog Queries of Boolean Constraint Databases, *International Journal of Algebra and Computation*, vol. 8, no. 5, pp. 553-574, 1998.

[10] P. Z. Revesz. Constraint Databases: A Survey, In: *Semantics in Databases*, Libkin, L. and Thalheim, B. editors, Springer-Verlag LNCS 1358, February 1998.

[11] P. Z. Revesz. Safe Datalog Queries with Linear Constraints. *Proc. Fourth International Conference on Principles and Practice of Constraint Programming*, Springer-Verlag LNCS 1520, 355-369, 1998.

[12] D. Toman, J. Chomicki, D.S. Rogers. Datalog with Integer Periodicity Constraints. *Proc. International Logic Programming Symposium*, 189–203, 1994.

[13] D. Toman, Foundations of Temporal Query Languages. Ph.D. thesis, Kansas State University, 1995.

[14] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, vols 1&2. Computer Science Press, 1989.