DATALOG WITH INTEGER PERIODICITY CONSTRAINTS*

DAVID TOMAN† AND JAN CHOMICKI‡

▷ In this paper we introduce a generalization of Datalog that operates on periodicity constraints over integers. We develop a closed-form bottom-up evaluation procedure for this class of constraints. We also develop a closed-form bottom-up query evaluation procedure for the class of periodic constraints combined with gap-order constraints. We provide complexity bounds for the query evaluation procedures. We extend this approach to allow combinations of classes of constraints over disjoint domain in the framework of Datalog. ◁

1. INTRODUCTION

Generalized databases [9, 10, 19] are infinite databases that can be represented using finite sets of *generalized* (or *constraint*) tuples. A number of query languages over such databases have been studied. The proposed query languages differ with respect to:

- the underlying inference mechanism (first order vs. deductive)
- the constraint language used.

In this paper we study generalized Datalog programs (function-free logic programs) that operate on constraint tuples in the place of ground atoms. This idea comes from constraint logic programming [7] and allows declarative specification of problems not solvable using the standard Datalog framework (e.g., reasoning about infinite sets of integers).

*A preliminary report on this work appeared in the proceedings of the 1994 International Logic Programming Symposium, Ithaca, NY.
†Department of Computer Science, University of Toronto Toronto, Ontario, Canada M5S 1A4 david@cs.toronto.edu.
‡Department of Computer Science, Monmouth University West Long Branch, NJ 07764, U.S.A. chomicki@moncol.monmouth.edu.

THE JOURNAL OF LOGIC PROGRAMMING

©Elsevier Science Publishing Co., Inc., 1993
655 Avenue of the Americas, New York, NY 10010 0743-1066/93/$3.50

Out work generally follows the conventions in [10]: In particular, constraints (constraint tuples[1]) are built out of a given set of *atomic constraints* over integers using finite conjunction and quantifier elimination (cf. Definitions 2.1 and 4.1). A *generalized relation* is a finite set of *quantifier free* constraints over the same set of variables that correspond to attributes of the relational schema of the relation.

Intuitively, a constraint represents the set of tuples that make the constraint true, when used as valuations. The meaning of a generalized relation is then the union of the sets of tuples corresponding to the individual constraints. In this way we can query infinite relations by performing all the necessary operations with respect to their finite encoding using constraints.

Example 1.1 (Example of Constraint Relations). The following examples show the use of constraints to represent finite and infinite relations over integers in a generalized database.

$P(1,2)$	tuple of arity 2 representing the set $\{(x,y) \mid x = 1 \wedge y = 2\}$
$R(x,y) :-\ x + 5 < y$	generalized tuple $\{(x,y) \mid x + 5 < y\}$ (order constraint)
$R(x,y) :-\ x \equiv_3 (y+1)$	$\{(x,y) \mid 3 \text{ divides } x - (y+1)\}$ (periodicity constraint)

The last two tuples represent infinite sets of integers. Constraint tuples can be combined using conjunction to produce tuples of higher arity in a natural way. Also, periodicity constraints over integers can be combined with order constraints, e.g.,

$$R(x) :-\ 0 < x, x \equiv_5 3, x < 1000000.$$

This constraint represents the finite set $\{3, 8, \ldots, 999998\}$. Nevertheless, the constraint representation is much more compact than the explicit representation of the same set and potentially leads to more efficient query evaluation.

In this paper we define a finite representation for periodic sets of integers. Periodic sets are useful for encoding and storing information about periodic activities in temporal databases, such as schedules, workflows, or experimental data. In addition, we show how such constraints can be combined with other classes of constraints over integers—gap-order constraints [12] and equality constraints over integers.

Applications of both periodicity and order constraints over integers can be found in several areas, including *temporal databases* where the time component is isomorphic to integers. The constraints are used to represent possibly infinite sets of time points [9, 19]. Periodicity and order constraints also serve to formulate general integrity constraints over temporal databases [8].

Example 1.2. A simple example of a Datalog program that uses integer constraints is a database of airline connections between cities (cf. Figure 1). A natural query over such a database is "Can I get from Paris to Toronto on Monday starting at 8 a.m. or later and arrive no later than 5 p.m.?" This example takes the advantage of both periodicity and gap-order constraints. Note that time units used in this example need to be converted to a common unit, in our case a minute; `1day` is then just a shorthand for 1440; similarly `Mon10:00` is 2040, and the notation `X = Mon10:00(mod 1day)` is a syntactic sugar for the constraint $X \equiv_{1440} 2040$. The representation of the inherently periodic information (like "flight every day") is

[1] In the constraint setting the distinction between an atomic constraint and a constraint tuple is blurred mainly because the atomic constraints do not have to be unary.

```
% Database of connections:
connection(paris, london, X, Y):-           % flight every day
        X = Mon10:00(mod 1day),
        Y = Mon11:30(mod 1day).
connection(paris, london, X, Y):-           % and weekdays
        Sun23:59 < Z, Z < Sat00:00,
        Z = Mon00:00(mod 1day),
        X = Z+14:30(mod 7days)
        Y = Z+16:00(mod 7days).
connection(london, toronto, X, Y):-         % twice a week
        X = Tue14:30(mod 7days),            % on Tuesday
        Y = Tue16:50(mod 7days).
connection(london, toronto, X, Y):-         % and
        X = Fri14:30(mod 7days),            % on Friday
        Y = Fri16:50(mod 7days).

% Change times for airports
changetime(london, X, Y) :- X+30<Y.         % 30' to change planes

% Planning of trips:
trip(From, To, Start, End) :-
        connection(From, To, Dtime, Atime),
        Start+15<Dtime,                     % 15' check in
        Atime+20<End.                       % 20' baggage claim
trip(From, To, Start, End) :-
        trip(From, Change, Start, Atime),
        trip(Change, To, Dtime, End),
        changetime(Change,Atime,Dtime).     % specific for airport
```

Figure 1. Example of Datalog query that uses integer constraints in an essential way.

very natural using periodicity constraints. The conditions of the type "you need at least 15 minutes to check in" are easily captured using gap-order constraints.

The main contribution of this paper is the definition of a closed-form bottom-up evaluation procedure for Datalog programs with integer periodicity and order constraints. This language can be used as a simple but very expressive query language over temporal databases. The evaluation for gap-order constraints is based on the results in [12, 13]. The proposed bottom-up evaluation procedure has *polynomial* data complexity[2]. Also, a general method for incorporating various classes of constraints into Datalog is studied.

1.1. Related Work

There are other proposals that introduce classes of infinite integer relations into various query languages and define closed-form bottom-up evaluation procedures for Datalog over these constraint classes: Datalog with (gap-)order constraints was introduced in [12]. In [9] relational calculus over generalized relational databases with order constraints and linear repeating points was studied (a linear repeating point has the form $\{c + kn | n \in Z\}$ for some fixed integers c and k and is just a different notation for a periodicity constraint). However, the proposed approach was limited to first order queries and thus simple inductive queries such as transitive closure could not be expressed. It is not clear how to adapt this approach to Datalog because of no obvious termination argument. Moreover, our approach tries to keep the different classes of constraints separate as long as possible (the only *meeting* points are the quantifier elimination and the consistency checking procedures). This arrangement provides a potentially more general framework, which could be adapted to combinations of other constraint languages over the integers.

Another approach comes from the area of temporal databases. In [4, 5] Datalog was extended with a limited use of the successor function symbol (the use is restricted to a single distinguished argument in each literal). In this way it is possible to represent infinite periodic sets of integers using Horn rules. But it is not clear how to add constraints to this language and maintain its computational properties. In particular, order constraints are not expressible in this language. Also, the unary successor symbol is used in the deductive layer on top of the database. Our approach allows a representation of the infinite relations to be directly stored in the database. Another extension of Datalog was proposed in [2]. This extension combines linear repeating points, order constraints, and unlimited use of successor function. The resulting language is very expressive, but the termination of query evaluation cannot be guaranteed.

There are also many proposals to extend a *first-order* query language (e.g., the relational calculus) with constraints [10]. However even the simplest extension beyond (gap-)order constraints, e.g., inclusion of the *linear arithmetic constraints* leads immediately to nontermination in the case of Datalog.

Summarizing: [12, 13] cannot handle periodicity constraints, [9] cannot handle recursion, [4, 5] cannot handle ordering, and [2] does not guarantee termination.

[2]Under few mild restrictions.

Our language is thus another step towards a tractable and expressive query language for temporal databases.

The rest of the paper is organized as follows. Section 2 gives the definition of a finite representation of periodicity constraints together with all the operations needed for the closed-form bottom-up evaluation. Section 3 describes the bottom-up evaluation procedure itself and proves its correctness and termination. The complexity of the evaluation procedure is also analyzed. Section 4 deals with the combination of the periodicity constraints with order constraints. Again, correctness, termination, and complexity results are presented. Section 5 shows how constraints over integers can be combined with constraints over other (disjoint) domains, e.g., uninterpreted database constants. The paper is concluded with a few open problems.

2. Periodicity Graphs

This section describes a graph-based representation of periodicity (congruence) constraints. The representation together with the operations defined on it serves as the basis for the closed-form bottom-up evaluation of Datalog programs with periodicity constraints.

Definition 2.1 (Periodicity Constraint). Let K be a finite set of natural numbers (modulo factors) and $\mathcal{C}_K$ a set of formulas defined inductively as follows:

1. $x_i \equiv_k (x_j + c)$ *where* x_i, x_j *are variables,* $k \in K$*, and* $c \in \{0, \ldots, k-1\}$
2. $x_i \equiv_k c$ *where* x_i *is a variable,* $k \in K$*, and* $c \in \{0, \ldots, k-1\}$
3. *if* $C_1, C_2 \in \mathcal{C}_K$ *then* $C_1 \wedge C_2 \in \mathcal{C}_K$
4. *if* $C \in \mathcal{C}_K$ *then* $\exists x.C \in \mathcal{C}_K$

Let C be an element of $\mathcal{C}_K$ and let X be the set of variables in C. Then we call C a periodicity constraint *over variables X.*

A constraint built only using rules (2) and (3) is called a simple periodicity constraint.

In the context of Datalog we can restrict our attention to the class of *simple periodicity* constraints without losing expressive power (cf. Section 4). This restriction allows a more efficient query evaluation procedure. However, the general theory developed in the section can be applied to other logic-based query languages over generalized databases with periodicity constraints, e.g., to conjunctive queries, where such simplification might not be possible.

Constraints in $\mathcal{C}_K$ are used in place of (ground) atoms during the bottom-up evaluation. They are directly represented using graphs:

Definition 2.2 (Periodicity Graph). Let $G = (N_G, E_G)$ be a complete undirected graph over $n+1$ nodes such that $N_G = \{0, x_1, \ldots, x_n\}$ and all edges $e \in E_G$ are labeled with a pair of positive integers $\alpha_G(e)$ and $\beta_G(e)$ where $0 \le \beta_G(e) < \alpha_G(e)$. We call G a periodicity graph *(of arity n) over variables $x_1, \ldots, x_n$.*

We define $\alpha_G(x, y) := \alpha_G((x, y))$ and $\beta_G(x, y) := \beta_G((x, y))$ for $x, y \in N_G$.

Disjunctions of periodicity constraints can be represented by a finite set of periodicity graphs.

Note that the periodicity constraints do not represent symmetric relations (with the exception of $\beta(x_i, x_j) = 0$). Without loss of generality we assume that the nodes of the graph are indexed by integers and thus can be ordered. This allows us to assume that the constraint between any two nodes of a given periodicity graph always relates the node with the lower index (the node 0 having index 0) to the node with the higher index, i.e., $x_i \equiv_{\alpha_G(x_i,x_j)} x_j + \beta_G(x_i, x_j)$ for $i < j$. To obtain the constraint for x_j, x_i (in this order) we use the following equivalence:

$$x_i \equiv_k x_j + c \iff x_j \equiv_k x_i + (k - c).$$

Thus it is sufficient to represent every constraint between any two variables by a single edge in the corresponding periodicity graph. The mapping ϑ used in this definition is merely a restriction of the given valuation to the free variables of C. To link the graph-based representation with the notion of constraint satisfaction we use the following definition:

Definition 2.3 (Assignment). Let G be a periodicity graph over $x_1, \ldots, x_n$ and

$$\vartheta : \{0, x_1, \ldots, x_n\} \to Z$$

be an assignment of integers to nodes $0, x_1, \ldots, x_n$ such that $\vartheta(0) = 0$. We say that ϑ satisfies *G ($\vartheta \models G$) if*

$$\begin{array}{lll} \forall x_i, x_j \in N_G : & \vartheta(x_i) \equiv_{\alpha_G(x_i,x_j)} \vartheta(x_j) + \beta_G(x_i, x_j) & \text{for } i < j \\ & \vartheta(x_i) \equiv_{\alpha_G(x_i,x_j)} \vartheta(x_j) - \beta_G(x_i, x_j) & \text{for } i > j. \end{array}$$

We call G a satisfiable periodicity graph *if there exists an assignment ϑ such that $\vartheta \models G$.*

We use the periodicity graphs to represent conjunctions of periodic constraints:

Definition 2.4. Let G be a periodicity graph over $x_1, \ldots, x_n$ and C a periodicity constraint over the same set of variables. We say that G is equivalent to *C if for all assignments ϑ*

$$\vartheta \models C \iff \vartheta \models G$$

The following definitions and lemmas show that every constraint in $\mathcal{C}_K$ can be represented by a periodicity graph of the appropriate arity. The graph is defined inductively with respect to the structure of the constraint using the following lemmas:

Lemma 2.5. Let $C = (x_i \equiv_k x_j + c)$ be a periodicity constraint and G a periodicity graph over $\{x_i, x_j\}$ where $\alpha_G(0, x_i) = \alpha_G(0, x_j) = 1$, $\beta_G(0, x_i) = \beta_G(0, x_j) = 0$, $\alpha_G(x_i, x_j) = k$, and $\beta_G(x_i, x_j) = c$. Then G is equivalent to C.

PROOF. Immediate from Definitions 2.2, 2.3, and 2.4.

Lemma 2.6. Let $C = (x_i \equiv_k c)$ be a periodicity constraint and G a periodicity graph over $\{x_i\}$ where $\alpha_G(0, x_i) = k$ and $\beta_G(0, x_i) = c$. Then G is equivalent to C.

PROOF. Immediate from Definitions 2.2, 2.3, and 2.4.

To combine constraints over the same variables we use following Proposition [6]:

Proposition 2.7 (Chinese Remainder Theorem). Let $k_1, k_2 \in N$, $0 \le c_1 < k_1$, and $0 \le c_2 < k_2$. If $\gcd(k_1, k_2)$ divides $|c_1 - c_2|$ then there exists a unique c such that

$$x \equiv_{k_1} (y + c_1) \wedge x \equiv_{k_2} (y + c_2) \iff x \equiv_{\mathrm{lcm}(k_1,k_2)} (y + c)$$

where $0 \le c < \mathrm{lcm}(k_1, k_2)$. Otherwise $x \equiv_{k_1} (y + c_1) \wedge x \equiv_{k_2} (y + c_2)$ is inconsistent.

This proposition allows us to form conjunctions of the periodicity constraints over two given sets of variables into a single constraint over the union of the variable sets or to show inconsistency of the original constraints. This idea is immediately reflected in the following algorithm that allows us to compute a *conjunction* of two periodicity graphs (more precisely, its output is a periodicity graph equivalent to the conjunction of the constraints represented by the input graphs):

Algorithm 2.8 (Conjunction). Let G_1 and G_2 be periodicity graphs over the sets of variables X and Y respectively. We construct the periodicity graph $G = G_1 \wedge G_2$ over $X \cup Y$ as follows:

1. *Let $N_G = N_{G_1} \cup N_{G_2}$ be nodes of graph G.*
2. *If for any edge $(x_i, x_j) \in E_{G_1} \cap E_{G_2}$ the condition*

$$\gcd(\alpha_{G_1}(x_i, x_j), \alpha_{G_2}(x_i, x_j)) \textit{ divides } |\beta_{G_1}(x_i, x_j) - \beta_{G_2}(x_i, x_j)|$$

   is violated, then the conjunction of G_1 and G_2 is not satisfiable and no graph is produced.
3. *Otherwise we label the edges $(x_i, x_j) \in E_G$ as follows*

$$\alpha_G(x_i, x_j) = \begin{cases} \mathrm{lcm}(\alpha_{G_1}(x_i, x_j), \alpha_{G_2}(x_i, x_j)) & (x_i, x_j) \in E_{G_1} \cap E_{G_2} \\ \alpha_{G_1}(x_i, x_j) & (x_i, x_j) \in E_{G_1} - E_{G_2} \\ \alpha_{G_2}(x_i, x_j) & (x_i, x_j) \in E_{G_2} - E_{G_1} \\ 1 & \textit{otherwise} \end{cases}$$

$$\beta_G(x_i, x_j) = \begin{cases} \gamma_{i,j} & (x_i, x_j) \in E_{G_1} \cap E_{G_2} \\ \beta_{G_1}(x_i, x_j) & (x_i, x_j) \in E_{G_1} - E_{G_2} \\ \beta_{G_2}(x_i, x_j) & (x_i, x_j) \in E_{G_2} - E_{G_1} \\ 0 & \textit{otherwise} \end{cases}$$

   where $0 \le \gamma_{i,j} < \alpha_G(x_i, x_j)$ is the unique solution of the equation

$$x_i \equiv_{\alpha_{G_1}(x_i,x_j)} x_j + \beta_{G_1}(x_i, x_j) \wedge x_i \equiv_{\alpha_{G_2}(x_i,x_j)} x_j + \beta_{G_2}(x_i, x_j)$$

   as in Proposition 2.7.

We use following lemma to show the correctness of this algorithm:

Lemma 2.9. Let C_1 and C_2 be periodicity constraints over the sets of variables X and Y respectively and G_1 and G_2 periodicity graphs equivalent to C_1 and C_2, respectively. If $C_1 \wedge C_2$ is satisfiable then there exists a periodicity graph $G = G_1 \wedge G_2$ over $X \cup Y$ equivalent to $C_1 \wedge C_2$.

PROOF. Let $\vartheta \models C_1 \wedge C_2$ be an arbitrary assignment. Then $\vartheta \models C_1$ and $\vartheta \models C_2$ by the assumption $\vartheta \models G_1$ and $\vartheta \models G_2$. Proposition 2.7 condition (2) of the

Algorithm 2.8 allows combining the constraints over the same pairs of nodes to a single and unique constraint. Because by the assumption the conjunction of the constraints is satisfiable, $\vartheta \models G_1 \wedge G_2$.

On the other hand, if $\vartheta \models G = G_1 \wedge G_2$, then obviously $\vartheta \models G_1$ and $\vartheta \models G_2$. By the assumption $\vartheta \models C_1$ and $\vartheta \models C_2$, and thus $\vartheta \models C_1 \wedge C_2$.

Lemmas 2.5, 2.6, and 2.9 allow us to construct a graph representation of an arbitrary periodicity constraint $C \in \mathcal{C}_K$ by induction on the structure of C:

Theorem 2.10. Let $C \in \mathcal{C}_K$ be a satisfiable periodicity constraint over $x_1, \ldots, x_n$. Then there exists a periodicity graph G over the same set of variables equivalent to this constraint.

PROOF. By induction on the structure of C. Base case follows from Lemma 2.5 and Lemma 2.6. The induction step follows from Lemma 2.9 and Lemma 2.12.

Note that equivalent constraints can be represented by different periodicity graphs. To avoid this problem, we introduce a notion of a *normal form* of periodicity graphs. The normalization algorithm also serves as the consistency checking algorithm: When checking whether or not an assignment of values to any two nodes of a periodicity graph satisfies all the constraints represented by the graph, we need to check all the paths between these two nodes in general. To avoid this problem we introduce a *normal form* for the periodicity graphs, where only a single edge needs to be checked for satisfaction.

Definition 2.11 (Normal Form). Let G be a periodicity graph. We say that G is in normal form *if for all $x_i, x_j, x_k \in N_G$*

$$(x_i \equiv_{\alpha_G(x_i,x_j)} (x_j + \beta_G(x_i, x_j))) \Rightarrow$$
$$(x_i \equiv_{\gcd(\alpha_G(x_i,x_k),\alpha_G(x_k,x_j))} (x_j + (\beta_G(x_i, x_k) + \beta_G(x_k, x_j))))$$

where $\gcd(x, y)$ *is the greatest common divisor of x and y.*

Lemma 2.12. All periodicity graphs in normal form are satisfiable.

PROOF. Let G be a periodicity graph in normal form over nodes $x_1, \ldots, x_k$. By induction on k:

Base case ($k = 0$): G contains a single node 0 and no edges. Clearly $\vartheta(0) = 0$ is an assignment that satisfies all the constraints.

Assume that all periodicity graphs in normal form over $k-1$ nodes are satisfiable. Thus for a subgraph G' of G with nodes $0, x_1, \ldots, x_{k-1}$ there is a satisfying assignment ϑ'. We extend this assignment to x_k as follows: we define the set X_k as follows:

$$\begin{array}{rl} X_k := \{x : & x \equiv_{\alpha_G(0,x_k)} \vartheta'(0) + \beta_G(0, x_k) \\ \wedge & x \equiv_{\alpha_G(x_1,x_k)} \vartheta'(x_1) + \beta_G(x_1, x_k) \\ \wedge & \ldots \\ \wedge & x \equiv_{\alpha_G(x_{k-1},x_k)} \vartheta'(x_{k-1}) + \beta_G(x_{k-1}, x_k)\}. \end{array}$$

This set is nonempty: if it were empty, some of the constraints defining X_k would have to be inconsistent and this would violate the assumption that G is in normal form. Now we let

$$\vartheta(x) = \begin{cases} \vartheta'(x) & x \in \{0, x_1, \ldots, x_{k-1}\} \\ y & x = x_k \wedge y \in X_k \end{cases}$$

Clearly $\vartheta \models G$ and thus G is satisfiable.

Definition 2.13. Periodicity graphs G_1 and G_2 are equivalent *if $\vartheta \models G_1 \iff \vartheta \models G_2$ for all assignments ϑ.*

To check satisfiability of a periodicity graph we use the following lemma:

Lemma 2.14. A periodicity graph is satisfiable if and only if it has an equivalent normal form.

PROOF. Let G' be a periodicity graph in normal form equivalent to G. Then G is satisfiable by Lemma 2.12. On the other hand, if G is a satisfiable periodicity graph then there is an assignment ϑ such that it satisfies every simple path between any two nodes of G. Because G is a finite graph, there are only finitely many simple paths between any two nodes, each of which is finite itself. Along every such path we can compose the constraints into a single constraint over the endpoints of the path. A conjunction of the resulting constraints must be satisfiable (this follows from G being satisfiable) and gives a assignment for the normal form of G.

Algorithm 2.15 (Periodicity Graph Normalization). Let $G = (N_G, E_G)$ be an arbitrary periodicity graph. The following algorithm computes the normal form for any satisfiable periodicity graph and fails for unsatisfiable periodicity graphs.

```
repeat
  for x_i, x_j ∈ N_G do
    for x_k ∈ N_G − {0, x_i, x_j} do
      M := lcm(α_G(x_i, x_j), gcd(α_G(x_i, x_k), α_G(x_k, x_j)))
      if M divides |β_G(x_i, x_j) − (β_G(x_i, x_k) + β_G(x_k, x_j))|
      then
        α_G(x_i, x_j) := M
        β_G(x_i, x_j) := γ
      else
        no graph can be produced and the algorithm fails
while α_G(x_i, x_j) changes for some i, j
```

where γ is the unique solution of the the pair of congruences $x_i \equiv_{\alpha_G(x_i,x_j)} x_j + \beta_G(x_i, x_j)$ and $x_i \equiv_{\gcd(\alpha_G(x_i,x_k),\alpha_G(x_k,x_j))} x_j + \beta_G(x_i, x_k) + \beta_G(x_k, x_j)$ as in Proposition 2.7.

For a closed-form bottom-up evaluation, two additional operations on the periodicity graphs are needed. The first one is *quantifier elimination* (constraint projection):

Algorithm 2.16 (Projection). Let G be a periodicity graph over a set of variables X and $x \in X$. We construct the periodicity graph $G' = \pi_x(G)$ over the set of variables $X - \{x\}$ as follows:

1. *Let G'' be the normal form of G.*

2. *Let G' be $(N_{G''} - \{x\}, E_{G''} - \{(x,y) | y \in E_{G''}\})$ and the labeling of the edges in G' be a restriction of the labeling in G''.*

Lemma 2.17. Let C be a satisfiable periodicity constraint over a set of variables X, $x \in X$, and G a periodicity graph equivalent to C. Then the periodicity graph $G' = \pi_x(G)$ over $X - \{x\}$ is equivalent to $\exists x.C$.

PROOF. Let ϑ be arbitrary such that $\vartheta \models C$. Clearly, $\vartheta \models G$ as well. Then obviously $\vartheta \models \exists x.C$ iff $\vartheta \models G'$.

Note that the 'iff' is preserved because the graph G'' is normalized, i.e., all the paths between any two nodes are 'implied' by the edge between these two nodes. In the case of the *simple* periodicity constraints, step 1. in Algorithm 2.16 can be omitted.

The last operation on the representation of periodicity constraints is *subsumption checking*. This operation is used in place of duplicate elimination used in ground Datalog:

Definition 2.18 (Subsumption). Let G_1 and G_2 be periodicity graphs over the same set of variables. We say that G_1 subsumes *G_2 if for all assignments $\vartheta : \vartheta \models G_2$ implies $\vartheta \models G_1$.*

Lemma 2.19. Let G_1 and G_2 be normal periodicity graphs over the same set of variables. Then G_1 subsumes *G_2 if for every edge (x,y)*

1. *$\alpha_{G_1}(x,y)$ divides $\alpha_{G_2}(x,y)$ and*
2. *$\beta_{G_1}(x,y) = \beta_{G_2}(x,y) \bmod \alpha_{G_1}(x,y)$.*

PROOF. Immediate from Definitions 2.11 and 2.18.

During evaluation of a Datalog$^{\equiv Z}$ program we often need to rename the variables in a constraint. Similarly we need to "rename" nodes in a periodicity graph that represents such constraint.

Definition 2.20 (Renaming). Let $\theta : \{x_1, \ldots, x_n\} \to \{v_1, \ldots, v_n\}$ be a renaming function and G a periodicity graph over $\{x_1, \ldots, x_n\}$. Let G' be a periodicity graph over $\{v_1, \ldots, v_n\}$ such that

$$\forall 0 \le i < j \le n : \quad \alpha_{G'}(x_i\theta, x_j\theta) := \alpha_G(x_i, x_j)$$

$$\beta_{G'}(x_i\theta, x_j\theta) := \begin{cases} \beta_G(x_i, x_j) & \text{for } x_i\theta < x_j\theta \\ \alpha_G(x_i, x_j) - \beta_G(x_i, x_j) & \text{for } x_i\theta > x_j\theta \end{cases}$$

where $v_i < v_j$ if $i < j$. We denote G' by $G\theta$.

Lemma 2.21. Let G be a periodicity graph equivalent to the periodicity constraint C and θ a renaming function. Then $G\theta$ is equivalent to $C\theta$.

PROOF. Immediate from the definition of periodicity graphs.

3. DATALOG WITH PERIODICITY CONSTRAINTS

The previous section introduced a finite representation of periodicity constraints. This representation is used in the bottom-up evaluation of Datalog$^{\equiv Z}$ programs—periodicity graphs serve as the elements of the computed interpretation. Datalog$^{\equiv Z}$ programs are defined naturally as follows:

*Definition 3.1 (*Datalog$^{\equiv Z}$ *program). An* atom *is a predicate symbol with distinct variables as arguments. A* Datalog$^{\equiv Z}$ program *is a finite set of clauses of the form*

$$A \leftarrow C, B_1, \ldots, B_k$$

where A and $B_1, \ldots, B_k$ are atoms and C is a satisfiable periodicity constraint in $\mathcal{C}_K$. Moreover, we require that the head of the clause has the form $A(x_1, \ldots, x_k)$ where k is the arity of A.

Note that the usual Datalog constants are not allowed here. This is not a problem; the constants can be added using the technique from Section 5. The restriction on the head of the clause guarantees that all the results produced for this particular predicate symbol are over the same set of variables.

In a Datalog$^{\equiv Z}$ program (with constraints from $\mathcal{C}_K$) every integer constant can be modeled by its remainder class in the additive cyclic group with $\mathrm{lcm}\, K$ elements $Z_{\mathrm{lcm}\, K}$. Thus, all the operations on numbers are performed in a "modulo" arithmetic.

We define the interpretation of a Datalog$^{\equiv Z}$ program with respect to constraint interpretations:

Definition 3.2 (Named periodicity graph). Let P be a Datalog$^{\equiv Z}$ *program, R a predicate symbol that occurs in P of arity k, and G a satisfiable periodicity graph over the set of variables $\{x_1, \ldots, x_k\}$. We call the pair (R, G) a* named periodicity graph.

An $\equiv$-interpretation *is a set of named periodicity graphs. We extend the subsumption relation as follows: A named periodicity graph (A, G)* is subsumed by an $\equiv$-interpretation *I if G is subsumed by a periodicity graph G' for some $(A, G') \in I$.*

Named periodicity graphs take the role of ground atoms in the standard bottom-up evaluation of Datalog:

Definition 3.3 ($\mathrm{TP}_\equiv$*). Let P be a* Datalog$^{\equiv Z}$ *program. Let I be a set of named periodicity graphs* (an $\equiv$-interpretation). *We define*

$$\begin{aligned}\mathrm{TP}_\equiv(I) = \{(A, \pi_{FV(A)}(G)) :\ & A \leftarrow C, B_1, \ldots, B_m \in P,\\ & (B_i, G_i) \in I \text{ for all } 0 < i \leq m,\\ & G \text{ is the normal form of } G_1\theta_1 \wedge \ldots \wedge G_m\theta_m \wedge G_C,\\ & (A, \pi_{FV(A)}(G)) \text{ is not subsumed by } I\}\end{aligned}$$

where $FV(A)$ is the set of free variables in A and θ_i is the renaming from $\{x_1, \ldots, x_{|FV(G_i)|}\}$ to $FV(G_i)$ as in Definition 2.20.

Let $\mathrm{TP}^0_\equiv(I) = I$ and $\mathrm{TP}^i_\equiv(I) = \mathrm{TP}_\equiv(\mathrm{TP}^{i-1}_\equiv(I))$ for $i > 0$.

Note that the nodes of the periodicity graphs are "renamed" using the variable names in the associated subgoals of the clause; the conjunction operation is then performed with respect to those names (similarly to the natural join). The standard bottom-up evaluation algorithm remains the same as for ground tuples (i.e., all the modifications needed for the evaluation of *constraint* queries are encapsulated in the definition of the $\mathrm{TP}_\equiv$ operator).

Algorithm 3.4 (Naive Bottom-up Evaluation). Let P be a Datalog$^{\equiv Z}$ *program and I and J two variables over $\equiv$-interpretations.*

$I := \emptyset$;
repeat
    $J := I$;
    $I := \mathrm{TP}_{\equiv}(I)$
while $J \neq I$;
return I

The rest of this section gives the proofs of correctness and termination, and complexity bounds for this algorithm.

3.1. Correctness

We show that Algorithm 3.4 produces the same result as the standard fixed-point iteration of an immediate consequence operator over ground interpretations.

Definition 3.5 (Immediate consequence operator over ground interpretations). Let I be a ground interpretation and P a Datalog$^{\equiv Z}$ *program. We define*

$$\mathrm{TP}(I) = \{A\theta : A \leftarrow D, B_1, \ldots, B_k \in P, \theta \models D \textit{ and } B_i\theta \in I\}$$

where θ is a valuation.

We use the following definition to relate the $\equiv$-interpretations to the ground interpretations:

Definition 3.6. Let I be a ground interpretation and $I_{\equiv}$ an $\equiv$-interpretation. We define a relation

$$I \sim I_{\equiv} \textit{ iff } I = \{A\theta : (A, G) \in I_{\equiv} \textit{ and } \theta \models G\}$$

for G a periodicity graph and θ a valuation.

The relation $\sim$ defines the meaning (semantics) of the $\equiv$-interpretations. The ground interpretation I can be viewed as a set of *unrestricted* relations [10]. To show that the bottom-up evaluation on $\equiv$-interpretations gives the same result (with respect to $\sim$) as the fixed-point iteration of the TP, we first show that a single application of TP and $\mathrm{TP}_{\equiv}$ preserves $\sim$:

Lemma 3.7. Let I be a ground interpretation and $I_{\equiv}$ an $\equiv$-interpretation, such that $I \sim I_{\equiv}$. Then $\mathrm{TP}(I) \sim \mathrm{TP}_{\equiv}(I_{\equiv})$.

PROOF. Let $A\theta \in \mathrm{TP}(I)$. Then there is a clause $A \leftarrow C, B_1, \ldots, B_k \in P$ such that $\theta \models C$ and $B_i\theta \in I$. By the assumption there are $(B_i, G_i) \in I_{\equiv}$ such that $\theta \models G_i$. Thus $G = \pi_{FV(A)}(G_1 \wedge \ldots \wedge G_k \wedge G_C)$ exists and $(A, G) \in \mathrm{TP}(I_{\equiv})$.

On the other hand, if $(A, G) \in \mathrm{TP}(I_{\equiv})$, then there is a clause $A \leftarrow C, B_1, \ldots, B_k$ in P such that $(B_i, G_i) \in I_{\equiv}$ and $G_1 \wedge \ldots \wedge G_k \wedge G_C$ is satisfiable. Let θ be the satisfying valuation. Then by the assumption $B_i\theta \in I$ and by the definition of TP $A\theta \in \mathrm{TP}(I)$.

Lemma 3.8. The immediate consequence operator $\mathrm{TP}_{\equiv}$ *is monotonic and continuous (preserves suprema of directed chains).*

PROOF. Immediate from Lemma 3.7 as TP is monotonic and continuous [11]

Now we prove the correctness of the closed form bottom-up evaluation algorithm by relating the iteration of the standard TP operator on ground interpretations to the iteration of $\mathrm{TP}_{\equiv}$. We compare the results obtained by iterating the TP and $\mathrm{TP}_{\equiv}$ operators with respect to the same $\mathrm{Datalog}^{\equiv Z}$ program P:

Theorem 3.9. For any $\mathrm{Datalog}^{\equiv Z}$ *program* P *and any predicate symbol* R *in* P

$$R(c_1,\ldots,c_k) \in \mathrm{TP}^{\omega}(\emptyset) \iff (R,G) \in \mathrm{TP}^{\omega}_{\equiv}(\emptyset) \wedge [c_1/x_1,\ldots,c_k/x_k] \models G.$$

PROOF. By simultaneous induction on TP and $\mathrm{TP}_{\equiv}$. The base case holds trivially and the induction step follows from Lemma 3.7.

Note that the number of iterations of TP and $\mathrm{TP}_{\equiv}$ is the same. However, the TP operator would have to operate on infinite ground interpretations.

This arrangement also shows how other evaluation procedures based on the constraint operations or the TP operator can be utilized for constraint query evaluation, e.g., the semi-naive bottom-up evaluation [22], or the SLG resolution [18, 21].

3.2. Termination

We show termination of an arbitrary $\mathrm{Datalog}^{\equiv Z}$ program by showing that the set of all possible periodicity graphs of given arity is finite for any given $\mathrm{Datalog}^{\equiv Z}$ program.

Definition 3.10 (Upper bound). Let P *be a* $\mathrm{Datalog}^{\equiv Z}$ *program,* $\{C_1,\ldots,C_n\}$ *a set of all constraints in* P *(including the input database), and* k_i *the modulo factor in the constraint* C_i*. Then we say that* $\alpha(P) = \mathrm{lcm}\{k_i | 0 < i \leq n\}$ *is the maximal modulo factor for* P*.*

We show that for a given $\mathrm{Datalog}^{\equiv Z}$ program (with respect to a fixed set $\mathcal{C}_K$) the labels of all the edges of the periodicity graphs are bounded by a constant. This is shown for the periodicity constraints first:

Lemma 3.11. Let C *be a satisfiable periodicity constraint that occurs in a* $\mathrm{Datalog}^{\equiv Z}$ *program* P *and* G *a periodicity graph equivalent to* C*. Then for all edges* $e \in E_G : \alpha_G(e) \leq \alpha(P)$*.*

PROOF. By structural induction on C. For an atomic C the claim holds trivially; for $C = C_1 \wedge C_2$, the claim follows from Lemma 2.9, and for $C = \exists x.C_1$, the claim follows from Lemma 2.17.

The application of $\mathrm{TP}_{\equiv}$ also preserves the bound:

Lemma 3.12. Let P *be a* $\mathrm{Datalog}^{\equiv Z}$ *program,* $I \geq 0$*, and* $(A,G) \in \mathrm{TP}^{i}_{\equiv}(\emptyset)$*. Then for all edges* $e \in E_G : \alpha_G(e) \leq \alpha(P)$*.*

PROOF. By induction on i. The claim holds trivially for $\emptyset$. Let $(A,G) \in \mathrm{TP}^{i+1}_{\equiv}(\emptyset)$. By definition of the $\mathrm{TP}_{\equiv}$ operator, (A,G) is the result of applying the $\mathrm{TP}_{\equiv}$ operator on the set $\mathrm{TP}^{i}_{\equiv}(\emptyset)$. The claim holds by IH for all periodicity graphs in $\mathrm{TP}^{i}_{\equiv}(\emptyset)$; it also holds for all satisfiable periodicity constraints by Lemma 3.11.

G is constructed using a clause in P, i.e., $G = \pi_{FV(A)}(G_1 \wedge \ldots \wedge G_m \wedge G_C)$. Thus, the claim holds for (A, G) by Lemma 2.9, Lemma 2.17, and the idempotence of the lcm operation.

Using the above two lemmas, we have:

Corollary 3.13. The bottom-up evaluation of any Datalog$^{\equiv Z}$ *terminates.*

3.3. Complexity of Bottom-up Evaluation for Datalog$^{\equiv Z}$

The number of applications of the $\mathrm{TP}_{\equiv}$ operator in Algorithm 3.4 can be bounded for each Datalog$^{\equiv Z}$ program as follows:

Lemma 3.14. Let P be a Datalog$^{\equiv Z}$ *program, R a predicate symbol in P, and a the arity of R. Then the number of different elements of* $\mathrm{TP}_{\equiv}^{\omega}(\emptyset)$ *of the form (R, G) is at most $\alpha(P)^{a(a+1)}$.*

PROOF. For every edge e in G we have only $\alpha(P)$ different labels $\alpha_G(e)$ by Lemma 3.12 and only $\alpha(P)$ different labels $\beta_G(e)$ by the definition of periodicity graphs (Definition 2.2). All the graph components of the elements $(R, G) \in \mathrm{TP}_{\equiv}^{\omega}(\emptyset)$ have exactly $\frac{a(a+1)}{2}$ edges.

The actual number of the periodicity graphs in the resulting model is usually smaller due to subsumption checking (Lemma 2.18). For *simple* constraints the bound is $\alpha(P)^{2a}$ as there are only a nontrivial edges in any of the periodicity graphs. Because the input database is represented by unit clauses in the Datalog$^{\equiv Z}$ program P, the data complexity is measured with respect to the number of clauses in P:

Theorem 3.15. Let P be a Datalog$^{\equiv Z}$ *program where all the atomic constraints are from $\mathcal{C}_K$ where n is the number of clauses in P (including facts). Then Algorithm 3.4 terminates in $O(n)$ steps.*

PROOF. Let a be the maximal arity of atoms in P, v the number of distinct variables in a single clause of P, p the number of predicate symbols in P, and m the maximal number of goals in any clause in P. An application of the $\mathrm{TP}_{\equiv}$ operator takes at most $mO(v^2) + O(v^4)$ steps for each clause in P and each of the $\alpha(P)^{a(a+1)}$ possible assignments of named periodicity graphs to goals in the body of the clause. The subsumption check takes another $a^2\alpha(P)^{a(a+1)}$ steps. Because there are at most $p\alpha(P)^{a(a+1)}$ different elements in the $\mathrm{TP}_{\equiv}^{\omega}(\emptyset)$ (by Lemma 3.14), the total time needed to compute the least model is at most:

$$n \cdot \left[\left(a^2 \alpha(P)^{a(a+1)} \right)^{m+1} \cdot \left(mO(v^2) + O(v^4) \right) \right] \cdot p\alpha(P)^{a(a+1)}.$$

Therefore the computation takes $O(n)$ time as a, m, p, v, and $\alpha(P)$ are constants for a given P with respect to $\mathcal{C}_K$ and $\alpha(P)$ is bounded by a constant that depends only on K.

For *simple* periodicity constraints the application of $\mathrm{TP}_{\equiv}$ takes only $mO(v)$ time. Note that the complexity of the subsumption checking does not affect the overall complexity of the bottom-up evaluation—it depends on the maximal arity of predicates but not on the size of the database.

The testing if a standard relational tuple belongs to the computed model can be also done in PTIME:

$\texttt{begin}(a_0)$. $\texttt{edge}(a_1,a_2)$. ... $\texttt{edge}(a_{n-1},a_n)$. $\texttt{end}(a_n)$.

$$\begin{array}{llllll}
\texttt{cong}(a_0, X) & \leftarrow & X \equiv_{p_0} 0 & \ldots & \texttt{cong}(a_0, X) & \leftarrow \; X \equiv_{p_0} (p_0 - 1) \\
\texttt{cong}(a_1, X) & \leftarrow & X \equiv_{p_1} 0 & \ldots & \texttt{cong}(a_1, X) & \leftarrow \; X \equiv_{p_1} (p_1 - 1) \\
\vdots & & & & \vdots & \\
\texttt{cong}(a_{n-1}, X) & \leftarrow & X \equiv_{p_{n-1}} 0 & \ldots & \texttt{cong}(a_{n-1}, X) & \leftarrow \; X \equiv_{p_{n-1}} (p_{n-1} - 1)
\end{array}$$

```
go(N,X):- end(N).
go(N,X):- cong(N,X), edge(N,M), go(M,X).
query(X):- start(N), go(N,X).
```

where $n > 0$, $p_0, \ldots, p_{n-1}$ are distinct prime numbers, and $a_0, \ldots, a_n$ distinct constants.

Figure 2. Datalog$^{\equiv Z}$ program with unrestricted number of modulo factors.

Theorem 3.16 (Tuple recognition). Let P be a Datalog$^{\equiv Z}$ *program where all the atomic constraints are from $\mathcal{C}_K$ and $A(c_1, \ldots, c_k)$ be a ground atom. Then $P \models A(c_1, \ldots, c_k)$ is decidable in $O(n)$.*

PROOF. The bottom-up evaluation procedure for P takes $O(n)$ time by Theorem 3.15, testing if A belongs to the $\mathrm{TP}^{\omega}_{\equiv}(\emptyset)$ takes constant time (with respect to the size of P) by Lemma 3.14.

In the rest of this section we show that the restriction to a fixed set K of the modulo factors is crucial for obtaining the polynomial bound[3]. We show an exponential lower bound for Datalog$^{\equiv Z}$ programs that do not meet this requirement. To show this bound we use the following proposition:

Proposition 3.17 (The cyclic primary decomposition theorem [3]). Any finite Abelian group is isomorphic to a finite product of cyclic groups of prime power orders, and the list of the prime power orders is unique up to permutation.

Using this proposition we construct a Datalog$^{\equiv Z}$ program that generates all elements of a sufficiently large group Z_P, where P will be at least exponential in the size of the Datalog$^{\equiv Z}$ program:

Example 3.18. Consider the Datalog$^{\equiv Z}$ program in Figure 2. Then the query $query(X)$ produces the set of constraint tuples

$$\{X \equiv_P i : 0 \leq i < P\}$$

where $P = p_0 \cdot p_1 \cdot \ldots \cdot p_{n-1}$. The size of this set is clearly exponential in the size of the input database as $n + \sum_{i=0}^{n-1} p_i < (n+1)p_{n-1}$. The fact that this program generates all the constraints follows immediately from Proposition 3.17 and the fact that the growth rate of the sequence of primes is approximately $n \log n$ [16, 17]; every element of Z_P is isomorphic to the product of the appropriate elements in the individual groups Z_{p_i}. These elements are represented by the constraints $X \equiv_{p_i} j$ and thus any particular element of the group Z_P is uniquely determined by the conjunction of such constraints over all p_i.

[3] It is not necessary to guarantee termination.

3.4. Negative Periodicity Constraints and Stratified Negation

One possible extension of Datalog$^{\equiv Z}$ is allowing negative periodicity constraints of the form $x \not\equiv_k y + c$. It is easy to see that such a constraint can be defined in Datalog$^{\equiv Z}$ using the following definition:

$$\begin{array}{lll} x \not\equiv_k (y+c) & :- & x \equiv_k (y + (c+1) \bmod k)) \\ x \not\equiv_k (y+c) & :- & x \equiv_k (y + (c+2) \bmod k)) \\ & \vdots & \\ x \not\equiv_k (y+c) & :- & x \equiv_k (y + (c+k-1) \bmod k) \end{array}$$

For a fixed K, the definitions represent a fixed set of clauses added to the original Datalog$^{\equiv Z}$ program. Therefore the data complexity results obtained in the previous section apply to this case as well.

Similarly we can introduce the *stratified negation* [22] to Datalog$^{\equiv Z}$ programs and preserve the data complexity bounds:

Definition 3.19. Let G be a periodicity graph over the set of variables X. Then we define

$$\neg^{\mathcal{C}}(G) = \{G^i_{x,y} : x, y \in N_G, x < y, 0 \le i < \alpha_G(x,y)\}$$

where $G^i_{x,y}$ *is a periodicity graph that represents a single atomic constraint* $x \equiv_{\alpha_G(x,y)} (y + (\beta(x,y) + i) \bmod \alpha_G(x,y))$.

Let I be a finite set named periodicity graphs and A an atom. We define

$$\neg^{\mathcal{C}}_A(I) = \{(A, \bigwedge^{\mathcal{C}} G'_i) : \text{ over all } (A, G_i) \in R \text{ such that } G'_i \in \neg^{\mathcal{C}}(G_i)\}$$

The $\neg^{\mathcal{C}}_A$ operation allows us to define the complement of a $\equiv$-interpretation with respect to a given atom. Clearly the above operations are well defined using the operations on periodicity graphs. Moreover

Lemma 3.20. Let θ be a valuation and I a $\equiv$-interpretation. Then

$$\begin{array}{ll} (i) & \exists (A,G) \in I.\theta \models G \vee \exists (A,G') \in \neg^{\mathcal{C}}_A(I).\theta \models G' \\ (ii) & \forall (A,G) \in I.\theta \not\models G \vee \forall (A,G') \in \neg^{\mathcal{C}}_A(I).\theta \not\models G' \end{array}$$

Moreover, $\neg^{\mathcal{C}}_A$ *preserves closure over* $\mathcal{C}_K$.

PROOF. Immediate from Definition of $\neg^{\mathcal{C}}_A$ and Lemma 2.9.

The bottom up evaluation of stratified Datalog$^{\equiv Z \neg}$ programs can be now defined by repeating the following two steps for every stratum of the given Datalog$^{\equiv Z \neg}$ program, starting from stratum 0:

1. apply Algorithm 3.4 on stratum j yielding an $\equiv$-interpretation I_j,
2. compute $\neg^{\mathcal{C}}_A(I_j)$ for every A appearing negatively in stratum $j+1$.

Correctness of this algorithm follows immediately from the correctness of Algorithm 3.4 and Lemma 3.20. In addition, because the $\neg^{\mathcal{C}}_A$ preserves closure over $\mathcal{C}_K$, the complexity analysis from Section 3.3 carries over to the case of stratified Datalog$^{\equiv Z \neg}$ programs. Note that the data complexity depends on the set of (prime) modulo factors present during the bottom-up evaluation of the program and that this set is not affected by the $\neg^{\mathcal{C}}_A$ operations. Therefore the complexity bounds remain the same as in the case of definite Datalog$^{\equiv Z}$.

3.5. Translation to Pure Datalog

In this section we investigate the following question: given a fixed set of modulo factors K, is it possible to express every Datalog$^{\equiv Z}$ program in standard Datalog? It is fairly easy to see that for every $k \in K$ there are only finitely many different periodicity constraints over two variables. However, we cannot merely substitute every pair of variables in the original clause by a variable holding the periodicity constraint between these two variables:

Example 3.21. Consider the following Datalog$^{\equiv Z}$ program (for simplicity we use only unary periodicity constraints):

$$a(X) \leftarrow b(X), c(X). \quad b(X) \leftarrow X \equiv_2 0. \quad c(X) \leftarrow X \equiv_3 0.$$

A naive translation to Datalog may try to use a ground representation of the periodicity constraints using pairs of values (similarly to a periodicity graph), yielding the following clauses:

$$a(X_k, X_c) \leftarrow b(X_k, X_c), c(X_k, X_c). \quad b(2,0). \quad c(3,0).$$

where X_k represents the modulo factor and X_c the remainder class. Now it is easy to see, that such a naive translation fails to produce $X \equiv_6 0$ (the least common multiple of 2 and 3).

The reason for the failure of the naive translation lies in the fact that in the constraint setting a value single variable in a Datalog$^{\equiv Z}$ clause can be *refined* by adding additional constraints. This is not possible in pure Datalog: once a variable is instantiated to a constant it can never change its value (as the only *constraints* in pure Datalog are of the form $x = a$).

In addition, unlike pure Datalog, Datalog$^{\equiv Z}$ allows constraints between two variables. Therefore, to perform a projection a proper quantifier elimination has to be performed (rather than simple dropping of the variable).

The way around these problems is to define the query processing on a *ground representation* of the periodicity constraints: Every atomic periodicity constraint $X_i \equiv_k X_j + c$ represented by a pair of integers (k, c) in the translated program[4]; the *true* constraint is represented by $(1, 0)$. Using this representation, for every clause

$$A \leftarrow C, B_1, \ldots, B_l.$$

in a given Datalog$^{\equiv Z}$ program we create a pure Datalog clause as follows:

1. For every atomic constraint $X_i \equiv_k X_j + c$ in C we add a goal $X_{ij}^0 = (k, c)$. We add $X_{ij}^0 = (1, 0)$ if no such constraint exists. The variable X_{ij}^0 represents the constraint between X_i and X_j, $i < j$; X_{0i}^0 stands for the constraint between 0 and X_i (the unary constraint on X_i).

2. For every atom $B_l(X_1, \ldots, X_k)$ we create an atom $B_l(X_{01}^{B_l}, X_{02}^{B_l}, \ldots, X_{kk}^{B_l})$ and add it to the body of the new clause.

[4] Technical we are allowed only atomic values in pure Datalog. However, it is easy to see that these pairs can be represented by a single integer as both the values come from a finite set.

3. If X_i and X_j, $i < j$ are free variables in B_l (or $i = 0$) we add the following conjunct to the body of the translated clause
$$and(X_{ij}^k, X_{ij}^{B_l}, X_{ij}^{k+1})$$
where *and* a relation holding the values obtained for all pairs of modulo factors in K using Proposition 2.7 and all the appropriate remainder classes; this table is finite because the set of modulo factors is fixed. The superscript k is the highest superscript used for X_{ij} in the generated clause so far (in this way we generate fresh names).

4. For all triples $i < k < j$ we add the conjuncts
$$trans(X_{il}^{k_1}, X_{lj}^{k_2}, X_{ij}^{k+1}), and(X_{ij}^k, X_{ij}^{k+1}, X_{ij}^{k+2})$$
where *trans* is a relation that captures compositions of two constraints similarly to the normalization algorithm and k, k_1, and k_2 are the highest superscripts used with the respective variables. This step is repeated t times where t is the number of variables in the clause.

5. The head of the resulting clause contains all the X_{ij}^k with the maximal k such that X_i and X_j, $i < j$, were in the original head (or $i = 0$).

Essentially, step 3 simulates computing the intersection of the individual periodicity graphs edge-by edge and step 4 represents the normalization step in the $\mathrm{TP}_{\equiv}$ operator. Step 3 can be avoided by converting all constraints in the given Datalog$^{\equiv Z}$ program to a common modulo factor $\operatorname{lcm} K$ (using disjunctions where necessary). However, this way we may end with a disjunction of (ground representations of) constraints rather than with a single constraint with a smaller modulo factor. Step 4 can be avoided by allowing only simple periodicity constraints (we see in the next section that this does not reduce the expressive power of the language).

4. Combining Classes of Constraints over Integers

In this section we show how periodicity constraints can be combined with other classes of constraints over integers in the framework of Datalog.

4.1. Gap-order Constraints

First we combine the constraint language developed so far with Datalog$^{<Z}$ (Datalog with gap-order constraints [12, 13]).

Definition 4.1 (Gap-order constraint). Let u and l be integers, c a nonnegative integer, and $x, y, \ldots$ be variables over integers. Then a finite conjunction of formulas of the form $l < x$, $x < u$, and $x + c < y$ is called a gap-order constraint *(tuple).*

Conjunctions of gap-order constraints can be efficiently represented using *gap graphs* [12] (directed acyclic graphs where nodes represent variables and the lower and the upper bounds of constraints, and directed labeled edges represent gaps—the minimal integer distances between nodes). This representation also supports all the necessary operations for closed-form bottom-up evaluation [12].

We show that the combination of these two approaches still has a closed-form evaluation procedure and the complexity bound does not increase.

Definition 4.2 (Datalog$^{\equiv_Z,<_Z}$ *programs). Let P be a finite set of clauses of the form*

$$A \leftarrow C_1, C_2, B_1, \ldots, B_k$$

where $A, B_1, \ldots, B_k$ are atoms, C_1 is a satisfiable periodicity constraint in $\mathcal{C}_K$, and C_2 is a satisfiable gap-order constraint. Then P is a Datalog$^{\equiv_Z,<_Z}$ *program.*

4.1.1. Complexity of Consistency Checking Consistency (satisfiability) checking of periodicity graphs and gap-order graphs (separately) can be done in polynomial time with respect to the size of the graph (cf. Section 2 and [12, 13]). However,

Theorem 4.3. Consistency checking of a conjunction of a periodicity graph with a gap graph is NP-complete.

PROOF. By reduction of 3SAT to satisfiability of conjunctions of constraints of the form $x \equiv_k (y+c)$ and $x+c < y$. Let S be an instance of 3SAT over variables $x_1, \ldots, x_k$ and clauses $c_1, \ldots, c_n$. We encode S by a quantifier-free conjunction of periodicity and gap-order constraints as follows:

Encoding of variables x_i: For every variable x_i we create a formula

$$\delta(x_i) = (z_i \equiv_2 z_i^* + 1) \wedge (-1 < z_i < 2) \wedge (-1 < z_i^* < 2)$$

Encoding of clauses c_i: Let $c_i = l_i^1 \vee l_i^2 \vee l_i^3$. We define a formula

$$\gamma(c_i) = (y_i \equiv_2 t_i^1) \wedge (y_i \equiv_3 t_i^2) \wedge (y_i \equiv_5 t_i^3) \wedge (0 < y_i < 30)$$

where $t_i^j = z_k$ if $l_i^j = x_k$ and $t_i^j = z_k^*$ if $l_i^j = \neg x_k$ for $0 < j \leq 3$.

The whole instance S is then reduced to the formula

$$\varphi(S) = \bigwedge_{0<i\leq k} \delta(x_i) \wedge \bigwedge_{0<i\leq n} \gamma(c_i)$$

It is easy to see that a satisfying assignment of values to the free variables of $\varphi(S)$ yields a satisfying assignment to S, where x_i is true if and only if $z_i = 1$. Clearly, $\varphi(S)$ can be represented as a conjunction of a periodicity graph with a gap graph.

On the other hand:

Theorem 4.4. Consistency checking of a conjunction of a periodicity graph constructed from simple periodicity constraints only and a gap graph can be decided in PTIME.

PROOF. By generalization of the consistency checking procedure given in [12]; the algorithm computes the length of every path from the lower bound to the upper bound of the gap-graph by adding the gap sizes on the edges of the path. Moreover, for every node along the path, the length of the partial path from the lower bound to this node is rounded up with respect to the periodicity constraint associated with this node. The graph is consistent iff the length of every path is shorter than the difference between the lower and upper bounds in the gap-graph. Figure 3 gives an algorithm for consistency checking of such conjunctions.

```
consistent(G,H) =
  if acyclic(G) then
    for every path l -e0-> x1 -e1-> ... -e(n-1)-> xn -en-> u do
      p := l + α_G(e0)
      for i := 1 to n do
        p := ⌈p⌉_xi
        p := p + α_G(ei)
      if p > u then
        return false
    return true
  else
    return false
```

where $\alpha_G(e)$ is the value of the gap associated with the edge E in G, l and u are the lower and upped bounds of G, and $\lceil p \rceil_{x_i}$ is the *rounding* up to the smallest integer p' greater than or equal to p and satisfying the constraint $p' \equiv_{\alpha_H(x_i)} \beta_H(x_i)$. Searching through all the paths from l to u can be efficiently achieved using a dynamic programming technique.

Figure 3. Consistency checking for conjunctions of simple periodicity and gap constraints.

Limiting the periodicity constraints to the form $x \equiv_k c$ does not reduce the expressive power of the query language. In the case of Datalog, all the constraints of the form $x \equiv_k (y+c)$ can be defined using the following Datalog$^{\equiv Z}$ clauses:

$$
\begin{array}{llll}
x \equiv_k (y+c) & :- & x \equiv_k 0 & \wedge \quad y \equiv_k c \\
x \equiv_k (y+c) & :- & x \equiv_k 1 & \wedge \quad y \equiv_k (c+1) \\
 & \vdots & & \\
x \equiv_k (y+c) & :- & x \equiv_k (k-1) & \wedge \quad y \equiv_k (c-1)
\end{array}
$$

The disadvantage of this solution is that we need to store more rules in the database.

Moreover, the overall complexity of the bottom-up evaluation is exponential in the *arity* of predicate symbols in the program. Thus even using a NP-complete consistency checking procedure does not have an impact on the overall data complexity. However, we consider only simple periodicity constraints in the rest of the paper, for which satisfiability checking is in PTIME.

4.1.2. Bottom-up Evaluation. Now we show that while we cannot compose the two classes over integers immediately, we can *reuse* a large portion of the algorithms developed so far:

1. we can reuse the constraint conjunction and subsumption operations for the individual classes, but
2. we have to develop a new constraint projection operation and a new satisfiability check.

Definition 4.5. A $(\equiv, <)$-interpretation is a set of triples (A, G, H) where A is a predicate symbol, G is a periodicity graph, and H is a gap graph, respectively. A, G, and H must have the same arity.

In the definition of the consequence operator $TP_{\equiv,<}$ we can reuse the conjunction and subsumption procedures defined for the component classes of constraints. We can also use the consistency check from Theorem 4.4. However, the most complex operation—the quantifier elimination—has to take care of the interactions between the periodicity constraints and the gap-order constraints [25]:

Example 4.6. Assume that we want to eliminate quantifier $\exists y$ from the constraint:

$$(\exists y)(x + c_1 < y \wedge y + c_2 < z \wedge y \equiv_k d)$$

Clearly we cannot replace it simply by $x + c_1 + c_2 + 1 < z$ as in the case of gap-order constraints: we need to take into account the periodicity constraint $y \equiv_k d$, i.e., we need to make sure that there is at least one integer of the form $\{d + nk\}$ between $x + c_1$ and $z - c_2$. Thus, the equivalent quantifier-free formula is

$$\begin{array}{ccc} (x + c_1 + 1) \equiv_k d & \wedge & x + c_1 + c_2 + 1 < z \quad \vee \\ (x + c_1 + 2) \equiv_k d & \wedge & x + c_1 + c_2 + 2 < z \quad \vee \\ & \vdots & \\ (x + c_1 + k) \equiv_k d & \wedge & x + c_1 + c_2 + k < z \end{array}$$

It is easy to see that the variable y was successfully eliminated and the resulting constraint is a disjunction of conjunctions of periodicity and gap-order constraints.

This idea is used for the projection operation needed in the $TP_{\equiv,<}$ operator; a similar idea was used in [9]; our procedure is simpler due to a different representation of constraints, especially we can omit the normalization (in the sense of [9]) of constraint tuples[5].

Algorithm 4.7 (Projection). Let G be a periodicity graph, H a gap-graph, y a node in both G and H, $x \in N_H$ such that $(x, y) \in E_H$ is labeled with c (i.e., $x+c < y$), $z_1, \ldots, z_l \in N_H$ such that $(y, z_j) \in E_H$ is labeled with c_j for $0 < j \leq l$, and $y \equiv_k d \in G$. We form a set of graphs (G_i, H_i) for $0 < i \leq k$ by modifying G and H as follows:

1. delete the edge (x, y) ,
2. insert edges (x, z_j) with the label $c + c_j + i$,
3. add the constraint $x \equiv_k (d - c - i)$,
4. check the resulting pair (G_i, H_i) for consistency and discard inconsistent pairs.

We apply this transformation recursively on each of the consistent pairs (G_i, H_i) until all the edges ending in y are eliminated from the gap graphs. Then we remove node y and all incident edges from all the resulting pairs of graphs.

Lemma 4.8. Let $\{(G_i, H_i) | i \in I\}$ be the set of graphs produced by Algorithm 4.7 from the pair (G, H) with respect to a variable y. Then for all valuations θ:

$$\theta \models \bigvee_{i \in I} (G_i, H_i) \iff \exists a \in Z. \theta[a/y] \models (G, H)$$

[5] The normalization in [9] is different from Definition 2.11 and requires the *whole relations* to be normalized.

```
qe(y,(G, H)) =
  if indegree(y) = 0
    then
      return {(G − {y}, H − {y})}
    else
      let x be a node such that (x, y) ∈ E_H
      and l be the number of nodes, such that (y, z_i) ∈ E_H in
        for all 0 < i ≤ l do
          create a gap graph H_i such that E_{H_i} = E_H − {(x, y)} ∪ {(x, z_i)}
            where α_{H_i}(x, z_i) := α_H(x, y) + α_H(y, z_i) + i
          create a simple periodicity graph G_i such that
            G_i := G ∧ G_{x ≡_k (α_G(y) − α_H(x,y) − i)}
        return ⋃_i (if (G_i, H_i) is consistent then qe(y,(G_i, H_i)) else {})
```

Figure 4. Pseudocode for Algorithm 4.7.

PROOF. By induction on indegree(y) (cf. Figure 5). If indegree(y) $= 0$ in G, then clearly for any valuation θ, $\theta \models (G - \{y\}, H - \{y\})$ if and only if $\theta[a/y] \models (G, H)$ for a sufficiently small $a \in Z$. Such an a must exist as there is no lower bound on the variable y in G.

For indegree(y) $= k$ we select a node x such that $(x, y) \in G$. One iteration of the qe procedure removes this edge from G and adds edges from x to all successors of y in G with the appropriate labels (cf. Example 4.6, Algorithm 4.7, and [25]). Thus the resulting set of graphs is equivalent to the original graph and indegree(y) $= k - 1$. Application of the inductive hypothesis on each of these graphs gives us the desired conclusion.

Thus, Algorithm 4.7 defines the projection operation for periodicity graphs combined with gap graphs. To eliminate all the existential quantifiers we apply this algorithm recursively to its own output until all the existential quantifiers are eliminated from the original pair of gap- and periodicity graphs.

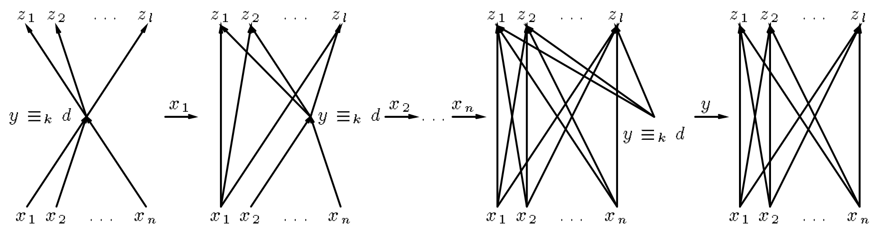
The projection operation produces a possibly fairly large set of graphs from a single graph. This is often considered a space complexity problem. However, in the framework of Datalog it is not true: even in the standard Datalog, each relation is represented as a collection of its tuples. In our case we still manage to represent often infinite sets of tuples by a single element.

The consequence operator is now defined in the usual way:

Definition 4.9. Let P be a Datalog$^{\equiv Z,<Z}$ *program and I an $\equiv, <$-interpretation.*

$$
\begin{aligned}
\mathrm{TP}_{\equiv,<}(I) = \{(A, G, H) : \; & A \leftarrow C_1, C_2, B_1, \ldots, B_m \in P \\
& (B_i, G_i, H_i) \in I \textit{ for all } 0 < i \leq m \\
& G' = G_1 \wedge \ldots \wedge G_m \wedge G_{C_1} \\
& H' = H_1 \wedge \ldots \wedge H_m \wedge H_{C_2} \\
& (G, H) \in \pi_{FV(A)}(G', H') \textit{ not subsumed by } I\}
\end{aligned}
$$

where G_{C_1} is a periodicity graph corresponding to C_1 and H_{C_2} is a gap graph corresponding to C_2.



The transitions labeled by x_i correspond to a recursive call of qe(y,(G, H)) after processing the edge (x_i, y) and the last step corresponds to the final removal of node y in the base step of the algorithm. Note that in every recursive call we produce up to k different graphs; the figure depicts only the shape of the generated graphs, not the actual labels.

Figure 5. Algorithm 4.7 applied to node y.

We omitted the renaming functions in the definition of $\mathrm{TP}_{\equiv,<}$ as for both gap-order graph and simple periodicity graphs the functions are trivial. We can show that the bottom-up evaluation of a Datalog$^{\equiv Z,<Z}$ program using Algorithm 3.4 (using $\mathrm{TP}_{\equiv,<}$ instead $\mathrm{TP}_{\equiv}$) is correct with respect to the standard bottom-up evaluation:

Theorem 4.10. For any Datalog$^{\equiv Z,<Z}$ *program P and any predicate symbol R in P*

$$R(c_1,\ldots,c_k) \in \mathrm{TP}^{\omega}(\emptyset) \iff (R,G,H) \in \mathrm{TP}^{\omega}_{\equiv,<}(\emptyset) \wedge [c_1/x_1,\ldots,c_k/x_k] \models G \\ \wedge [c_1/x_1,\ldots,c_k/x_k] \models H$$

PROOF. By simultaneous induction on the iterations of TP and $\mathrm{TP}_{\equiv,<}$; similarly to the proof of Theorem 3.9.

To show termination of Datalog$^{\equiv Z,<Z}$ we use the termination results shown for for Datalog$^{\equiv Z}$ and Datalog$^{<Z}$:

Definition 4.11. Let I be a $(\equiv,<)$-interpretation. We define

$$\pi_{\equiv}(I) = \{(A,G) : (A,G,H) \in I\} \text{ and } \pi_{<}(I) = \{(A,H) : (A,G,H) \in I\}.$$

We use the *projections* of the interpretation to bound the total number of iterations of the $\mathrm{TP}_{\equiv,<}$ operator.

Lemma 4.12. Let P be a Datalog$^{\equiv Z,<Z}$ *program. Then*

$$\pi_{\equiv}(\mathrm{TP}^{i}_{\equiv,<}(\emptyset)) \subseteq \mathrm{TP}^{i}_{\equiv}(\emptyset) \text{ and } \pi_{<}(\mathrm{TP}^{i}_{\equiv,<}(\emptyset)) \subseteq \mathrm{TP}^{i}_{<}(\emptyset)$$

for all $0 \le i$, where $\mathrm{TP}_{\equiv}$ *(*$\mathrm{TP}_{<}$*) is the immediate consequence operator with respect to the program P where all the gap-order (periodicity) constraints have been removed.*

PROOF. By induction on i.

Corollary 4.13. The bottom-up evaluation terminates for any Datalog$^{\equiv Z,<Z}$ *program.*

Combination of results in [12, 14] shows DEXPTIME-completeness of Datalog$^{<Z}$ (again, a data-complexity result). The addition of periodicity constraints does not affect this result.

To show a polynomial bound for the tuple recognition procedure for Datalog$^{\equiv Z,<Z}$ programs we use the same technique: the tuple recognition runs in PTIME (follows from Theorem 3.15 and the complexity of the TEST Algorithm [12])[6].

4.1.3. Negation. Adding stratified negation to Datalog$^{<Z}$ is nontrivial: [12] shows that Datalog$^{<Z,\neg}$ with full stratified negation is Turing-complete (and thus termination of queries cannot be guaranteed), [15] defines a syntactic restriction on stratified Datalog$^{<Z,\neg}$ programs where queries are computable (but nonelementary). The second approach can be immediately combined with unary periodicity constraints to obtain a safe version of stratified Datalog$^{\equiv Z,<Z,\neg}$. The addition of periodicity constraints does not affect the query evaluation complexity.

4.2. Equality Constraints

So far we have not mentioned the most natural class of constraints on integers—the equality constraints, which are needed to include the "standard" Datalog over integers. There are two kinds of equality constraints we want to consider:

1. $x = c$, where x is a variable and c is an integer constant. This kind of constraints can be handled directly using the gap-order constraints:

$$(x = d) \iff (d - 1 < x < d + 1)$$

2. $x = y$, where both x and y are variables. In [12] this kind of constraints is handled by adding extra information to the gap-graphs. In the case of Datalog$^{\equiv Z}$ it can be done in the same way; in all cases we need some mechanism to represent a single constraint of the form $x = y$.

For constraint classes closed under conjunction we can use following lemma:

Lemma 4.14. Let φ be a constraint where x and y are free variables. Then

$$\varphi \wedge (x = y) \iff \varphi \wedge \varphi[x/y, y/x]$$

After applying this lemma we can eliminate either x or y using the quantifier elimination procedure described in the previous section. If $x = y$ is the only constraint we can just ignore it.

5. CLASSES OF CONSTRAINTS WITH DISJOINT DOMAINS

The previous section suggests that we can integrate various classes of constraints into Datalog. The main difficulty of doing so is the design of the quantifier elimination procedure (projection). This becomes much simpler when we combine classes

[6] The degree of the polynomial is higher than in standard Datalog—$O(a^2)$ for atoms of arity a (exactly $(a+1)(a+2)/2$ for Datalog$^{<Z}$ and $2a+2$ for Datalog$^{\equiv Z}$). This is because the values are assigned to *pairs* of attributes rather than to single attributes. In both cases the *exponentiated* expression is a constant with respect to the size of the program.

of constraints with disjoint domains (like integers and uninterpreted constants). Then because of the 'constraint' approach, the evaluation of each of the classes is independent of the other classes. The only meeting point is consistency checking or tuple membership checking (the tuple membership checking may be considerably less complex than consistency checking).

Definition 5.1 (Datalog$^{\mathcal{S}_1,\dots,\mathcal{S}_n}$). Let $\mathcal{S}_i$ be a class of constraints for $0 < i \leq n$. We say that a finite set of clauses of the form

$$A \leftarrow C_1, \dots, C_n, B_1, \dots, B_m$$

where A and $B_1, \dots, B_m$ are predicate symbols, and C_i is a satisfiable constraint in $\mathcal{S}_i$, is Datalog$^{\mathcal{S}_1,\dots,\mathcal{S}_n}$ program.

Note that each of the classes $\mathcal{S}_i$ contains the trivial constraint *true*.

Definition 5.2. Let $\mathcal{S}$ be a class of constraints, $\mathrm{TP}_{\mathcal{S}}$ be the consequence operator for Datalog$^{\mathcal{S}}$, and P be a Datalog$^{\mathcal{S}}$ program and C a clause in P. The relativized consequence operator *is defined as $\mathrm{TP}_{\mathcal{S}}$ with respect to the program $\{C\}$ and denoted $\mathrm{TP}^C_{\mathcal{S}}$.*

A $(\mathcal{S}_1, \dots, \mathcal{S}_n)$-interpretation is a set of $(n+1)$-tuples $(A, G_1, \dots, G_n)$ where A is a predicate symbol and G_i are finite representation of a constraint in $\mathcal{C}_i$. All G_i's have the same arity as A.

Definition 5.3. Let P be a Datalog$^{\mathcal{S}_1,\dots,\mathcal{S}_n}$ program, such that the domains of $\mathcal{S}_i$ and $\mathcal{S}_j$ are disjoint for all $i \neq j$. Let I be a $(\mathcal{S}_1, \dots, \mathcal{S}_n)$-interpretation. Let $\pi_{\mathcal{S}_j}(I) = \{(A, G_j) : (A, G_1, \dots, G_n) \in I\}$. Then let

$$\mathrm{TP}_{\mathcal{S}_1,\dots,\mathcal{S}_n}(I) = \{(A, G_1, \dots, G_n) : \begin{array}{l} A \leftarrow B_1, \dots, B_m, C_1, \dots, C_n \in P \\ G_i = \mathrm{TP}^{A \leftarrow B_1,\dots,B_m,C_i}_{\mathcal{S}_i}(\pi_{\mathcal{S}_i}(I)) \ \textit{exists}\ \} \end{array}$$

Note that G_i must exist for each $0 < i \leq n$. This may not always be the case (e.g., in the case of Datalog$^{\equiv Z,<Z}$ no atom is produced if the resulting gap-graph is inconsistent).

The correctness of the bottom-up evaluation of Datalog$^{\mathcal{S}_1,\dots,\mathcal{S}_n}$ is proven similarly to the Datalog$^{\equiv Z,<Z}$ case:

Theorem 5.4. For a Datalog$^{\mathcal{S}_1,\dots,\mathcal{S}_n}$ program P and a predicate symbol R in P

$$R(c_1, \dots, c_k) \in \mathrm{TP}^\omega(\emptyset) \iff \begin{array}{l} (R, G_1, \dots, G_n) \in \mathrm{TP}^\omega_{\mathcal{S}_1,\dots,\mathcal{S}_n}(\emptyset) \\ \wedge[c_1/x_1, \dots, c_k/x_k] \models G_i \ \textit{for all}\ 0 < i \leq n \end{array}$$

PROOF. By simultaneous induction on the number of iterations of TP and $\mathrm{TP}_{\mathcal{S}_1,\dots,\mathcal{S}_n}$ similarly to the proof of Theorem 3.9.

The termination of the bottom-up evaluation for Datalog$^{\mathcal{S}_1,\dots,\mathcal{S}_n}$ is proven by an extension of the technique used in Lemma 4.12:

Lemma 5.5. Let P be a Datalog$^{\mathcal{S}_1,\dots,\mathcal{S}_n}$ program. Then

$$\pi_{\mathcal{S}_j}(\mathrm{TP}^i_{\mathcal{S}_1,\dots,\mathcal{S}_n}(\emptyset)) \subseteq \mathrm{TP}^i_{\mathcal{S}_j}(\emptyset)$$

for all $0 < j \leq n$ and $0 \leq i$.

PROOF. Immediate from Definition 5.3.

Note that the $\mathrm{TP}^i_{\mathcal{S}_j}$ operator is applied on clauses of P projected to $\mathrm{Datalog}^{\mathcal{S}_j}$ (i.e., all constraint goals except C_j are deleted from the clauses of P).

Corollary 5.6. Let $\mathrm{TP}^\omega_{\mathcal{S}_j}$ *be finite for every* $0 < j \leq n$. *Then the* $\mathrm{TP}^\omega_{\mathcal{S}_1,\ldots,\mathcal{S}_n}(\emptyset)$ *is finite and thus the bottom-up evaluation of an arbitrary* $\mathit{Datalog}^{\mathcal{S}_1,\ldots,\mathcal{S}_n}$ *program terminates.*

PROOF. Immediate from Lemma 5.5.

We can use the standard bottom-up evaluation algorithm and the termination of all queries is guaranteed. Moreover, the product of the cardinalities of the $\mathrm{TP}^\omega_{\mathcal{S}_j}$ sets bounds the number of TP applications needed to reach the fixed point. Especially, if all the sets are polynomial in the size of the program then we have PTIME evaluation procedure. The tuple recognition procedure for $\mathrm{Datalog}^{\mathcal{S}_1,\ldots,\mathcal{S}_n}$ can be based on the combination of tuple recognition procedures for $\mathrm{Datalog}^{\mathcal{S}_j}$ in similar way as in Definition 5.3.

We use this definition to add the standard domain of "uninterpreted constants" to our framework of constraints.

Example 5.7. Let D be a flat domain of "uninterpreted constants" equipped with equality only. Then we can define language $\mathrm{Datalog}^{=D}$ as follows:

1. a =-interpretation is a set of tuples (A, R_A) where A is an atom and R_A is a corresponding A-tuple over D.
2. a conjunction of two atoms (A, R_A) and (B, R_B) is a tuple where fields of R_A are matched against fields of R_B with the same name,
3. a conjunction of atoms is consistent if the matching is successful,
4. a projection operation is simple removing a field from the tuple, and
5. (A, R_A) subsumes (A, S_A) iff $S_A = R_A$

Both the $\mathrm{TP}_{=D}$ operator and the bottom-up evaluation procedure are an immediate instance of Definition 5.3.

The bottom-up evaluation of $\mathrm{Datalog}^{=D}$ program is exactly the same as in standard Datalog. But this is not the only way of thinking about bottom-up evaluation in $\mathrm{Datalog}^{=D}$. We can also choose the =-interpretation to be a set of finite relations over D. Then conjunction becomes the natural join, projection becomes the relational algebra projection, and subsumption becomes the subset relation. Also consistency checking is easy: we simply look only for nonempty relations.

Now we can use Definition 5.3 to define the language $\mathrm{Datalog}^{=D,<Z,\equiv Z}$. The result is the language developed in the previous section enriched by the standard domain of uninterpreted constants. The evaluation procedure is again the standard bottom-up evaluation algorithm.

Note that we don't need a specialized consistency checking procedure over all the conjunctions of $= D, < Z$, and $\equiv Z$ constraints. The reason is that the domains D and Z are disjoint and thus we can check the consistency separately.

6. CONCLUSION

We have described bottom-up evaluation procedures for several versions of Datalog enhanced by constraints over the integers and provided a general way for combining various classes of such constraints into a single language. We have also shown complexity bounds for the bottom-up evaluation algorithm.

Below we list some further directions of research.

Expressiveness. In [1] the expressiveness of a number of deductive and constraint query languages is discussed. However, only monadic programs are considered there. It is interesting to see, whether the expressiveness of query languages defined in this section can also be formally characterized.

Efficient Implementation. Periodicity constraints define non-convex sets. In this respect they differ from most common constraint languages. [10] describes how to adapt interval management techniques for indexing in constraint databases. However, this approach works only for convex sets. Thus, periodicity constraints call for new storage management techniques.

BIBLIOGRAPHICAL COMMENTS

The work on the combination of Datalog rules with integer order constraints was pursued in [12, 13]. Integer based constraint queries have also been proposed in [9]. Most of the results about integers can be easily found in books on number theory, e.g., [6, 24], or [17] (if you read Polish). Other useful results about cyclic groups Z_n can be found in books on group theory and modern algebra in general, e.g., [3].

REFERENCES

1. Baudinet, M., Chomicki, J., Wolper, P. Temporal Deductive Databases. In *Temporal Databases: Theory, Design, And Implementation*, Tansel, A. et al. (editors). Benjamin Cummings 1993.
2. Baudinet, M., Niézette, M., Wolper P. On the Representation of Infinite Temporal Data and Queries. *10th ACM Symposium on Principles of Database Systems,* 280–290, 1991.
3. Birkhoff, G., MacLane, S. Algebra. The MacMillan Co., 1967.
4. Chomicki, J., Imieliński, T. Temporal Deductive Databases and Infinite Objects. Proc. *7th ACM Symposium on Principles of Database Systems,* 61–73, 1988.
5. Chomicki, J., Imieliński, T. Finite Representation of Infinite Query Answers, In *ACM Transactions on Database Systems*, 181–223, (18) 2, 1993.
6. Hardy, G. H. An Introduction to the Theory of Numbers. Oxford University Press, 1979.
7. Jaffar, J., Lassez, J. L. Constraint Logic Programming. Proc. *14th ACM Symposium on Principles of Programming Languages*, 111–119, 1987.
8. Jensen, C. S., Snodgrass, R. Temporal Specialization and Generalization. *IEEE Transactions on Knowledge and Data Engineering* 6(6): 954–974, 1993.
9. Kabanza, F., Stevenne, J-M., Wolper, P. Handling Infinite Temporal Data. *Journal of Computer and System Sciences*, (51) 1, 149–186, 1995.
10. Kanellakis, P. C., Kuper, G. M., Revesz, P.Z. Constraint Query Languages, *Journal of Computer and System Sciences*, (51) 1, 26–52, 1995.
11. Lloyd, J. Foundations of Logic Programming. Springer-Verlag, 1987.

12. Revesz, P. Z. A Closed Form Evaluation for Datalog Queries with Integer Order. Proc. *3rd International Conference on Database Theory*, 187–201, Springer-Verlag LNCS 470, 1990.
13. Revesz, P. Z. A Closed Form Evaluation for Datalog Queries with Integer (Gap)-Order Constraints. *Theoretical Computer Science*, vol. 116, no. 1, 117–149, 1993.
14. Revesz, P. Z. Datalog Queries of Set Constraint Databases. In *Proc. 5th International Conference on Database Theory*, 423–438, 1995.
15. Revesz, P. Safe Stratified Datalog with Integer Order Programs. In *Proc. First International Conference on Constraint Programming*, Montanari U., Rossi F. eds., Springer-Verlag LNCS 976, Cassis, France, September 1995.
16. Shapiro, H. N. Introduction to the Theory of Numbers. John Wiley & Sons, 1983.
17. Sierpiński, W. Teoria Liczb, Part II, Warsaw, 1959.
18. Swift, T., Warren, D. S. Analysis of SLG-WAM Evaluation of Definite Programs. Proc. *1994 International Logic Programming Symposium*, MIT Press, 1994, 219–235.
19. Tansel A., Clifford J., Gadia S., Jajodia S., Segev A., Snodgrass R. Temporal Databases. Theory, Design, and Implementation. Benjamin Cummings 1993.
20. Toman, D., Chomicki, J., Rogers D. S. Datalog with Integer Periodicity Constraints. Proc. *1994 International Logic Programming Symposium*, 189–203, MIT Press, 1994.
21. Toman, D. Memoing Evaluation for Constraint Extensions of Datalog. To appear in the International Journal on *Constraints*, special issue on Databases. 1997. An extended abstract appeared as Top-Down beats Bottom-Up for Constraint Based Extensions of Datalog. Proc. *1995 International Logic Programming Symposium*, 98–112, MIT Press, 1995.
22. Ullman J. D. Principles of Database and Knowledge-base Systems, Vol. 1,2. Computer Science Systems, 1989.
23. Vardi, M. Y. The Complexity of Relational Query Languages. In *ACM SIGACT Symposium on Theory of Computing,* 137–146, 1982.
24. Weil, A. Basic number theory. 3ed. Springer-Verlag, 1974.
25. Williams, H. P. Fourier–Motzkin Elimination Extension to Integer Programming Problems. In *Journal of Combinatorial Theory* (A) 21, 118-123, 1976.

Acknowledgment

The participation of David S. Rogers in the early stages of this research and the discussions with Peter Revesz are gratefully acknowledged. The authors would also like to thank one of the reviewers for suggesting the possibility of translating Datalog$^{\equiv Z}$ to pure Datalog.