A Closed Form Evaluation For Datalog Queries With Integer Order Constraints

Peter Z. Revesz

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-91-39
May 15, 1991

A CLOSED FORM EVALUATION FOR DATALOG QUERIES WITH INTEGER ORDER CONSTRAINTS*

(May 15, 1991)

Peter Z. Revesz[†]

Abstract: We provide a generalization of Datalog based on generalizing databases with the addition of integer order constraints to relational tuples. For Datalog queries with integer order constraints we show that there is a closed form evaluation. We also show that the tuple recognition problem can be done in PTIME in the size of the (generalized) database, assuming that the size of the constants in the query is logarithmic in the size of the database. Note that the absence of negation is critical, Datalog⁻ queries with integer order constraints can express any Turing computable function.

1 Introduction

In this paper we consider an extension of Datalog based on the notion of a constraint tuple. The important idea of a constraint tuple comes from constraint logic programming systems, e.g. CLP [14], Prolog III [4], and CHIP [8], and it generalizes the notion of a ground fact. This allows the declarative programming of new applications, including various combinatorial search problems (see [28] for a survey). Recently we extended this idea to database systems through the design of CQLs or *Constraint Query Languages* [17]. For example, in the relational database model $R(3,4)$ or $R(x,y)$ with $x = 3, y = 4$ is a tuple of arity 2. In our framework, $R(x,y)$ with $x = y, x < 2$ is a generalized tuple of arity 2 and so is $R(x,y)$ with $x + y = 25$, where x and y are any integers satisfying these constraints. An important feature of generalized tuples of arity k is the description by a finite representation of a possibly infinite set of regular relational tuples with arity k.

A CQL can be considered to be a union of a database query language and a decidable logical theory together with a bottom-up evaluation in closed form. For a CQL to be feasible the bottom-up evaluation has to be efficient. A generally accepted measure of performance of a bottom-up evaluation is *data complexity*, which was introduced by

---

†Department of Computer Science, Brown University, Providence, RI.

Chandra and Harel [5], and by Vardi [29]. Data complexity measures the complexity of answering a fixed query in terms of the size of the database. The rationale behind data complexity is that the size of the database dominates the query size by several orders of magnitude for most applications.

It is well-known that any Datalog and any inflationary Datalog¬ [1, 20] query evaluated on traditional relational databases have PTIME data complexity. We show in [17] that any query in the language of inflationary Datalog¬ combined with the theory of rational order [9] or in the language of inflationary Datalog¬ combined with the theory of equality over an infinite set of constants can be evaluated bottom-up in PTIME data complexity. The latter case is also considered in [2, 18, 11, 24]. However, the case of Datalog combined with the theory of integer order [10] was left as an open problem. Here is a simple example of the type of queries that can be expressed in this language.

Example 1.1

$$T(x,y) :\!\!- T(x,z), T(z,y)$$

$$T(x,y) :\!\!- E(x,y), H(x,h), H(y,h'), h < h'$$

Assume that the input relation H is a function of "height" on the vertices of a binary input relation E of edges a graph. Then the query, which is expressed using Datalog with integer order constraints, finds in T the transitive closure of E using only "ascending" edges. Hence for any a and b vertices, $T(a,b)$ is an output of the query if and only if there is an uphill path of E edges from a to b. □

There are other attempts to combine some form of integer constraint solving with existing relational database languages. Recently Kabanza, Stevenne, and Wolper [15] examined a way of combining Relational Calculus with a limited form of recursion called *linear repeating points*. (Linear repeating points are points of the form $c + kn$, for each integer n and fixed integer constants c and k.) As [15] shows, for unary and binary predicates this gives exactly Presburger definability, which is a clear extension of Relational Calculus, but it still lacks expressibility of simple operations such as transitive closure. Another recent and related paper by Chomicki and Imielinski [7] builds directly on Datalog by restricting the application of the successor function to one argument (say always the first argument) of relations. This restriction disallows the order predicate, which cannot be expressed in this language. Hence, both of these results leave open the natural case of Datalog combined with the theory of integer order.

Our main difficulty in integrating relational database languages and integer order constraints was developing the appropriate quantifier elimination procedure for the positive existential subset of the theory of integer order. The quantifier elimination procedure that we develop can be used recursively to evaluate Datalog queries with integer order.

> This paper describes for Datalog with integer order queries a bottom-up evaluation method that terminates in closed form on any input generalized relational database (Theorem 3.15). The evaluation method always gives some

(possibly non-unique) output generalized relational database. This ouput database is equivalent to the unique least fixpoint model of the implicit unrestricted relational database input (Theorem 3.17).

In general the output of our evaluation procedure is not unique. Note that this does not cause any problems because the least fixpoint model of any query is a *unique* unrestricted relational database. The situation of having several possible generalized relational database outputs is only due to the fact that the same unrestricted relational database can be described finitely by several syntactically different generalized relational databases.

We also show that for any Datalog with integer order query and any relational tuple $A(t)$, testing whether $A(t)$ belongs to the (unrestricted) model of the query can be done in data complexity $O((u - l)^c + n^k)$ time, where c is some constant, n is the size of the database, k is the largest arity of a relation in P, and l is the smallest and u is the largest constant in either t or the database (Theorem 4.8). For the case of linear recursive programs the test is shown to be in NLOGSPACE data complexity (Theorem 4.9).

The existence of a closed form for Datalog with integer order is pleasantly surprising. It is easy to see that stratified Datalog¬ with integer order is undecidable as this constraint query language can express all Turing-computable functions (see Proposition 2.3). The key reason for the undecidability is the first-order expressibility of the successor function, which together with the recursive power of Datalog gives the expressibility of all μ-recursive functions.

Proposition 2.3 is worth comparing with the complexity results of Immerman [12] and Vardi [29]. They show that any query in any language with PTIME data complexity can be expressed in Datalog¬ with an integer order predicate, and conversely, any query in Datalog¬ with an integer order predicate has PTIME data complexity. Immerman and Vardi, however, use only traditional relational databases, whereas in Proposition 2.3 we use generalized relational databases.

An important technical contribution of this paper is the definition of a *gap-graph* on integer variables. Intuitively, k-variable gap-graphs partition the k-dimensional point space and capture the essential properties of each subset. A gap-graph is a graph with variables and two constants as vertices and some (undirected or directed) edges between distinct vertices together with a *gap-order labeling* for each edge. A gap-order is either $=$ on an undirected edge or $<_g$ on a directed edge for some nonnegative integer subscript g that denotes the minimum difference between the two ordered elements. (That is, the vertices can represent either integer constants or integer variables. In the latter case, the gap-order labels are meant to restrict the possible values that the variables can take.) While gap-graphs arise naturally, they potentially form an infinite partition of the point space (since there is no limit a priori on the gap values): the challenge is to show that a finite number of gap-graphs always suffice to describe any input and output generalized database. We show that true in a very general setting using a geometric argument.

We start by giving some definitions and propositions in Section 2. The closed form evaluation method for Datalog with integer order queries is described in Section 3. Also in Section 3, we prove that the method terminates on any input and computes a well-defined unique model for each query. The analysis of the tuple recognition problem for Datalog with integer order queries in general and for the special case of linear recursive queries is given in Section 4. Finally, open problems are listed in Section 5.

2   Basic Concepts

Our work will be a particular generalization of Codd's *relational database model* [3] and the theoretical toy language *Datalog*, which is built on fixpoint logic. We assume that the reader is familiar with these concepts and their relevance to databases. An introduction to these can be found for example in [16, 26]. In this section we will give only the necessary definitions for the new generalizations of relational databases and Datalog.

Generalized relational databases: Our database framework is set up as follows. Let $A(x_1, \ldots, x_k)$ be a relation symbol with arity k. Let $t \in Z^k$ be any sequence of k integers. (We denote by Z the set of integer numbers.) We call t a *tuple* and $A(t)$ a *relational tuple*. In the relational model database relations are composed of a finite number of relational tuples. An *integer order constraint* is of the form $u = v, u \neq v, u \leq v$, or $u < v$, where u and v are variables or constants. Variables range over the *intensional domain Z*. Constants and $=, <, \leq$ are interpreted as integer numbers and their ordering. Let $\phi(x_1, \ldots, x_k)$ be a conjunction of integer order constraints over distinct variables $x_1, \ldots, x_k$. We call ϕ a *constraint tuple*. We call an expression of the form $A(x_1, \ldots, x_k)$:— $\phi(x_1, \ldots, x_k)$ a *generalized relational tuple*, where $x_1, \ldots, x_k$ are distinct variables, and when there is no confusion about substitutions we write $A(\phi)$.

We view each generalized relational tuple $A(x_1, \ldots, x_k)$:— $\phi(x_1, \ldots, x_k)$ as a finite decription for a possibly infinite number of relational tuples $A(t_1), \ldots, A(t_n)$, where each $t_i \in Z^k$ and t_i satisfies ϕ, i.e. $t_i \models \phi$ in the standard sense. Therefore, we are dealing with special types of unrestricted (finite or infinite) relational databases. For each database and query program we call the finite set of constants appearing in them the *active domain D*.

Closed form bottom-up evaluation: We require that the evaluation of the query given an input generalized relation yield an output generalized relation which has the same type of constraints as the input generalized relation. This we call the *closed form* requirement. We make this requirement to enable a easy composition of queries.

We also require that the evaluation be bottom-up, i.e., that it evaluate the subgoals before the head of the rules. We make this requirement to allow the possibility of compiler and run-time optimizations. We know that many good optimization methods for relational database languages are based on bottom-up evaluation. For more about optimization and the importance of bottom-up evaluation see [16, 26].

Safety: Recall that an assumption in Codd's relational model is that relations (both

4

input and output) are always finite structures, that is, they are always composed of a finite number of relational tuples. This is called the *safety* requirement. Analogous to that requirement, in our extension of the relational database model, we require that relations be composed of a finite number of generalized relational tuples. We make this requirement to assure that the queries are always evaluable in finite time.

An important point to note here is that in the relational database model, to guarantee safety (i.e., that the output database is finite) only a restricted subset of relational calculus, namely *safe relational calculus* is allowed as a query language [3]. By contrast, our general evaluation method works for any Datalog with integer order query, that is, the evaluation method finds a finite generalized output database in finite time for any finite generalized input database. Let us now describe our query language.

Datalog with integer order constraints: The syntax is that of traditional Datalog where the bodies of rules can also contain a conjunction of integer order constraints. A *Datalog* program P with integer order, is a finite set of rules of form:

$$A_0 :- A_1, A_2, \ldots, A_l.$$

The expression A_0 (the rule *head*) must be an atomic formula of the form $R(x_1, \ldots, x_n)$, and the expressions $A_1, \ldots, A_l$ (the rule *body*) must be atomic formulas of the form $x_i = x_j$, $x_i \neq x_j$, $x_i < x_j$, $x_i \leq x_j$, or $R(x_1, \ldots, x_n)$, where R is some predicate symbol.

The semantics of a Datalog program P on a generalized database $r_1, \ldots, r_n$, (representing unrestricted relational database $\rho_1, \ldots, \rho_n$), is the least fixpoint of the monotone mapping defined by a first-order formula ϕ_P and $\rho_1, \ldots, \rho_n$. We explain ϕ_P by an example right below. This is as in the case without constraints, the only difference being the use of unrestricted relational databases [16, 23]. The following example appears in [17], except there the variables are interpreted to range over the rationals.

Example 2.1 Consider the Datalog with integer order query P:

$$R(x, y) :- R(x, z), R_0(z, y), x \leq y, y \leq z$$

$$R(x, y) :- R_0(x, y)$$

Let this query be applied to a generalized database r_0 representing the unrestricted relation ρ_0. Then ϕ_P is the following first-order formula, where R_0 is interpreted as ρ_0:

$$\phi_P(x, y) \equiv \psi(x, y; R) \equiv R_0(x, y) \lor \exists z (R(x, z) \land R_0(z, y) \land x \leq y \land y \leq z).$$

This formula defines a mapping from unrestricted relations ρ of arity 2 to unrestricted relations of arity 2. Here R is singled out because it is the predicate variable that is initialized to ρ and will receive the output of ϕ_P. This mapping is

$$\rho \longrightarrow \{a, b \in Z^2 | < Z, \delta, \rho_0, \rho > \models \phi_P(a, b)\}$$

where we denote by δ our interpretation of the symbols $=, \neq, <$, and $\leq$. This mapping is monotone with respect to set inclusion for ρ. By the Tarski fixpoint theorem it has a least fixpoint, which is the output of the query program applied to input r_0. $\square$

Datalog with integer gap-order constraints: The syntax is that of Datalog with integer order constraints with one addition. We allow in a rule body and in the generalized database atomic formulas called *gap-orders* of the form $x_i <_g x_j$ where $g \in N$. (We use N to denote the set of natural numbers.) We interpret this as a limited form of addition, namely as $x_i + g < x_j$. Otherwise the semantics is the same as for Datalog with integer order constraints. Let us illustrate this language also with an example.

Example 2.2 An example of Datalog with integer gap-order constraints is the following:

$$P(x, y, s_1, s_2) :\!- P(x, z, s_1, s_3), E(z, y, s_3, s_2)$$

$$P(x, y, s_1, s_2) :\!- E(x, y, s_1, s_2)$$

Suppose that the generalized database inputs are of the form

$$E(x, y, s_1, s_2) :\!- x = \text{city1}, y = \text{city2}, s_1 <_g s_2$$

Then the query expresses a variant of the shortest path problem. We can think of the database inputs as representing in the first two arguments by integer id numbers pairs of cities between which there is a direct path. We can think of the third and fourth arguments as representing by variables the endpoints of a stretched-out string and the gap between the endpoints representing the distance between the two cities.

Intuitively, these strings need to be concatenated to measure long distances. The query will do exactly that, i.e., concatenate the strings along all possible paths. Then it becomes easy to check whether there is a path of length l between any two given cities c_1 and c_2 by testing whether in the generalized output database there is a generalized relational tuple $P(x_1, x_2, x_3, x_4) :\!- \phi(x_1, x_2, x_3, x_4)$ such that $t \models \phi(x_1, x_2, x_3, x_4)$, where $t = (c_1, c_2, 0, l)$. $\square$

Remark: The type of test in Example 2.2 is called a *tuple recognition* test, i.e., we want to find whether a regular relational tuple t is in the generalized output database. We know that l is the shortest path between cities c_1 and c_2 if the recognition test succeeds for tuple $t = (c_1, c_2, 0, l)$ and fails for tuple $t' = (c_1, c_2, 0, l - 1)$. Recognition tests like the one in Example 2.2 motivate our analysis of the problem in Section 4.

Generalized Data Complexity: The way Chandra and Harel [5] and Vardi [29] define data complexity is based on tuple recognition. That is, they actually phrase the query evaluation problem as a decision problem. They define for a fixed query Q a language $L_Q = \{(t, d) : t \in Q(d)\}$. That is, the language consists of the set of strings which are pairs of a regular relational tuple t and an input database d (also written as a string of regular relational tuples) such that t is in the output database in a complete evaluation of Q on d. In Section 4 we use a modified definition of data complexity to allow d to be a generalized database. We define the language as $L_Q = \{(t, d) : t \models g \land g \in Q(d)\}$, where t is a regular relational tuple that satisfies some generalized relational tuple g that is in the generalized output database in our evaluation of Q on d.

Stratified Datalog¬ with integer order constraints: The syntax is that of Datalog with integer order constraints with one addition. We allow in a rule body expressions of the form $\neg R(x_1, \ldots, x_n)$, where R is some predicate symbol. We give the language stratified semantics [6]. Recall that we call a relation an *intensional database* (or IDB for short) relation if it is defined in terms of other database relations, i.e., the relational predicate symbol occurs in the head of one or more rules of the program. The stratified semantics introduced by Chandra and Harel means that we group IDB predicates (and rules in which they occur as head) into strata, or layers, and evaluate each layer in sequence. The only restriction that we make is that an IDB predicate should not occur negated before the computation passed its layer.

We give the following simple proposition to illustrate that dealing with integer order can be hard and to put our results on Datalog with integer order into a better context.

Proposition 2.3 Any Turing-computable function is expressible by a query of Stratified Datalog¬ with integer order constraints.

Proof: Since the class of Turing-computable and the class of μ-recursive functions are equivalent (see [19, 21]), it is enough to show that all μ-recursive functions can be expressed. Recall from [21] (see Definition 5.5.2 in [21]) that a function is μ-recursive if and only if it can be obtained from the initial functions of zero, projection, and successor by the operations of composition, primitive recursion, and unbounded minimalization applied to regular functions. (See for the other basic definitions also [21]. We repeat here for clarity only the definition of *regular*. A $(k+1)$-place function g is *regular* if and only if, for every $\overline{n} \in N^k$, there is an m such that $g(\overline{n}, m) = 0$.)

In our Datalog¬ simulation we use the convention that the function values of the initial and all built-up functions are in the last argument of the predicates with the same name that the functions have. The zero function is immediate. We do it by simply letting $zero(0)$ be a database fact. The projection functions are also straightforward. For each ith k-place projection function with $1 \le i \le k$ we let

$$project_i(x_1, \ldots, x_n, x_{n+1}) :\!\!- x_{n+1} = x_i$$

be database facts. The successor function can be expressed as follows:

$$succ(x, y) :\!\!- x < y, \neg distant(x, y)$$

$$distant(x, y) :\!\!- x < z < y$$

In the above the bottom rule must be done before the top rule by stratification. Together the two rules say that y is the successor of x, if y is greater than x and there is no integer z between x and y.

Let $l > 0$ and $k \ge 0$ and let g be an l-place function and f and $h_1, \ldots, h_l$ be k-place functions. Then the composition of f obtained from g and $h_1, \ldots, h_l$ can be simulated as follows:

$$f(\overline{n}, y) :\!\!- g(\overline{x}, y), h_1(\overline{n}, x_1), \ldots, h_l(\overline{n}, x_l)$$

Primitive recursion is also straightforward:

$$f(\overline{n}, 0, y) :\!\!- g(\overline{n}, y)$$

$$f(\overline{n}, m', y) :\!\!- h(\overline{n}, m, x, y), f(\overline{n}, m, x), succ(m, m')$$

For the unbounded minimalization stratified negation is again necessary. As in the definition of successor, the bottom rule is evaluated first.

$$f(\overline{n}, m) :\!\!- g(\overline{n}, m, 0), \neg smaller(m', m)$$

$$smaller(m', m) :\!\!- g(\overline{n}, m', 0), m' < m$$

Finally, we note that the depth of the stratification need not be large, because of a theorem of Kleene [19] which states that only one application of unbounded minimalization is sufficient to express any μ-recursive function. □

Remark: It follows from [17] that stratified Datalog¬ with rational order can be evaluated in PTIME in the size of any generalized extensional database. Clearly, the difficulty comes from having the integers as the domain not from negation. In the theory of rational order we cannot express many things that we can express in the theory of integer order [9], for example we cannot express successor ($+1$).

3 A Closed Form Evaluation for Datalog with Integer Order

In this section we show that for Datalog with integer order programs and any generalized input database there is a closed form evaluation. This evaluation strategy will be similar to the naive evaluation algorithm used for Datalog only. That is, at each step all possible rule applications are considered with all possible substitutions of database facts for subgoals. The complication results from the database facts being generalized relational tuples.

Our main goal is to prove that there is an evaluation algorithm for which the number of steps necessary is finite and that the algorithm computes a well-defined, unique output for each query. (For complexity results see Section 4). We will proceed as follows.

In Section 3.1 we simplify the problem by transforming database queries with integer order constraints into a special form, where each generalized tuple is represented as a *gap-graph* (Definitions 3.1-3.2 and Lemma 3.3). Second, we give some basic notions about gap-graphs the most important of which is *consistency* (Definitions 3.4-3.6) and show a simple way to test whether a gap-graph is consistent (Lemma 3.7).

In Section 3.2, we define *shortcut, merge, subsume,* and *extend* as the four basic operations on gap-graphs (Definitions 3.8-10). We also show in several lemmas the semantic correctness of these operators. Essentially that means that the operators are consistency preserving (Lemmas 3.11-3.13).

In Section 3.3 we define generalized rule applications and our evaluation method in terms of the four basic operations on gap-graphs. We prove in Theorem 3.15 that the

evaluation method always terminates (the key lemma is 3.14). Finally, we prove in Theorem 3.17 that the evaluation method always yields the desired unique fixpoint model (defined in Definition 3.16).

3.1 Gap-Graphs

For this section we let D to be any fixed finite set of integers. We start with the basic definitions of *gap-orders* and *gap-graphs*.

Definition 3.1 Let x and y be any two integer variables. Given some assignment to the variables, a *gap-order* $x <_g y$ for some gap-value $g \in N$ holds if and only if $x + g < y$ holds in the given assignment. A *gap-order* $x = y$ holds if and only if x and y are equal in the given assignment. $\square$

Definition 3.2 Let $x_1, \ldots, x_n$ be a set of integer variables, and let l and u be any elements of $D \cup \{+\infty, -\infty\}$ such that $l < u$. Then a *gap-graph* is any graph that has $n + 2$ vertices labeled $x_1, \ldots, x_n$ and l and u and has between any pair of distinct vertices at most one undirected edge labeled by $=$ or one directed edge labeled by a gap-order $<_g$ for some $g \in N$. $\square$

In the following we will always assume when talking about edges that an edge labeled by $=$ is undirected and an edge labeled by a $<_g$ for some $g \in N$ is directed. The direction in the latter case is necessary only to make it clear which vertex is less than the other. We will assume that if a directed edge from vertex v to another vertex u has label $<_g$ on it, then $v <_g u$ is the gap-order constraint that is represented within the gap-graph.

Two examples of gap-graphs are shown in Figure 3(a) and (b). The first one has both undirected and directed edges, while the second has only directed edges.

Gap-graphs provide an alternative representation of generalized tuples which are conjunctions of integer order constraints. That is spelled out in Lemma 3.3. Lemma 3.3 helps to transform the problem of evaluating a Datalog with integer order query into a graph problem. We can assume later that the generalized input database has only gap-graphs as generalized tuples.

Lemma 3.3 Any (satisfiable) conjunction of integer order constraints can be represented as a finite (nonempty) disjunction of gap-graphs.

Proof: See Appendix. $\square$

The following three definitions describe some concepts that are related to gap-graphs and are used repeatedly throughout Section 3.

Definition 3.4 For each gap-graph G, we call the graph obtained by deleting all gap-order labels in G the *underlying graph* of G. For each graph G, we call the graph obtained
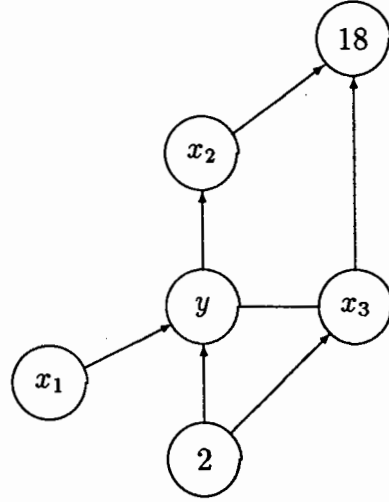
Figure 1: Example of underlying graph

by merging all equal vertices and deleting all edges with = labels the *compact graph* of G.
□

For example, Figure 1 is the underlying graph and Figure 2 is the compact graph of the gap-graph in Figure 3(a). Note that compact graphs have only directed edges. Also note that a compact graph may have multiple labels for vertices and multiple edges between vertices. Using Definition 3.4 simplifies our proofs when only the properties captured by the underlying graphs or the compact graphs are important. For instance, it is simpler to talk about path lengths within compact graphs than within gap-graphs.

Definition 3.5 A directed edge form vertex v to another vertex w in a gap-graph we denote as (v, w). A chain of directed edges $(v, w_1), (w_1, w_2), \ldots, (w_{n-1}, w_n)$ we call a *path*. If $v \equiv w_n$, then we call the path a *cycle*. The *length* of the path from v to w_n is the sum of the gap values in labels of the directed edges plus $(n - 1)$. If there is no directed edge leading to a vertex, then the vertex is called a *leaf*. A gap-graph is *acyclic* if and only if its compact graph is acyclic. □

For example, $(x_1, y), (y, x_2), (x_2, 18)$ is a path of the gap-graph in Figure 3(a). The length of the path is 9. The leaves of the gap-graph are x_1 and 2. The gap-graph is acyclic, because its compact graph in Figure 2 is acyclic.

Definition 3.6 A gap-graph G is *consistent* if and only if there is an integer assignment $\mathcal{A}$ to the variables in the vertices that satisfies all the gap-order labels on the edges. We denote this as $\mathcal{A} \models G$. □

By examining the algorithm within Lemma 3.3, it is easy to see that any integer tuple t satisfies the conjunction of integer order constraints if and only if t is a *consistent*
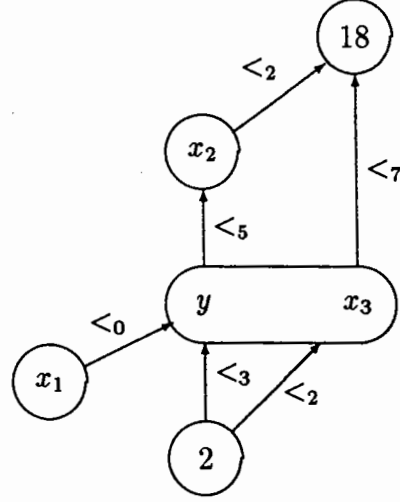
10

Figure 2: Example of compact graph

assignment to at least one of the gap-graphs produced by the algorithm. If the conjunction of integer order constraints is unsatisfiable, then the algorithm within Lemma 3.3 does not produce any gap-graph. This makes precise what we mean by alternative representation. It is also clear that alternative representation is semantically equivalent in the usual sense to the conjunctive integer order formula.

Therefore, the notion of consistency of a gap-graph is naturally linked with the notion of satisfiability of a conjunctive integer order formula. The advantage of using the former is that it can be easily checked whether a gap-graph is consistent. That check is described in Lemma 3.7.

Lemma 3.7 Let G be a gap-graph with vertices $v_1, \ldots, v_n, l, u$, where $l, u \in D \cup \{+\infty, -\infty\}$ and $l < u$. When $l, u \in D$, G is consistent if and only if it is acyclic and in the compact graph of G the longest path from l to u is less than $(u - l)$ and there is no path from u to l. When $l = -\infty$ or $u = +\infty$, G is consistent if and only if it is acyclic and there is no directed edge $(v, l) = (v, -\infty)$ or $(u, v) = (+\infty, v)$.

Proof: See Appendix. □

3.2   The Operations on Gap-Graphs

In this section we describe the four basic operations on gap-graphs. These operations are called *shortcut* (Definition 3.8), *merge* (Definition 3.9), *subsume* (Definition 3.10), and *extend*.

The four operations are defined on gap-graphs that serve as generalized tuples. The definitions of these operations can be broadened to generalized relations that are composed
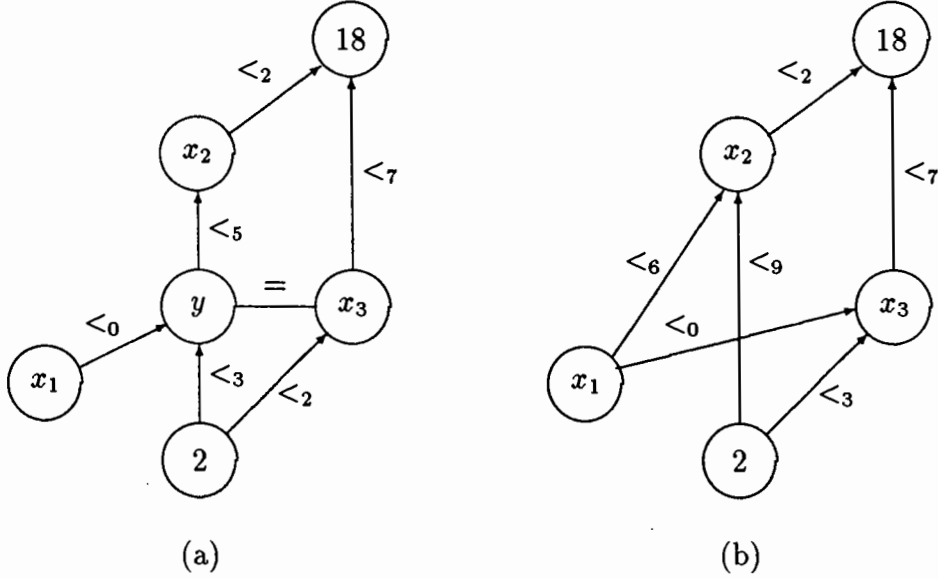
11

Figure 3: Example of shortcut of a gap-graph

of a finite number of gap-graphs. Intuitively, in that case the shortcut would replace the projection, the subsume would replace the difference, and the extend and merge together would replace the natural join operation in relational algebra.

Definition 3.8 Let G be a gap-graph with vertices $y, v_1, \ldots, v_n, l, u$, where $l, u \in D \cup \{+\infty, -\infty\}$ and $l < u$. Then a *shortcut* operation over vertex y transforms G into a gap-graph G' with vertices $v_1, \ldots, v_n$ and $v_{n+1} = l$ and $v_{n+2} = u$ as follows.

First, for each $0 < i, j, \leq n + 2$ do the following.

If $v_i = y$ and $y = v_j$, then add $v_i = v_j$ as an undirected edge to G.
If $v_i = y$ and $y <_g v_j$, then add $v_i <_g v_j$ as a directed edge to G.
If $v_i <_g y$ and $y = v_j$, then add $v_i <_g v_j$ as a directed edge to G.
If $v_i <_{g_1} y$ and $y <_{g_2} v_j$, then add $v_i <_{g_1+g_2+1} v_j$ as a directed edge to G.

Second, for each g and h if $v_i <_g v_j$ and $v_i <_h v_j$ and $g < h$, then delete first edge. If more than one edge remains between any two vertices, then the shortcut operation fails, returns an error message, and it does not produce a shortcut graph as output. Otherwise, also delete y and all edges incident on y. □

The intuition behind the shortcut operation is fairly straightforward. We want to eliminate a vertex y. Just erasing y and the edges incident on y is not enough, because they imply gap-order constraints about other vertices and that information would be lost. We need to explicitly preserve that information. The first part of the operation does exactly that. It is easy to see that it tests all possible cases in which two edges incident on y can imply a new gap-order constraint. Of course the listing makes use of symmetry. Also, note that the case $v_i <_{g_1} y$ and $y >_{g_2} v_j$ is left out, because these two edges do not imply a new gap-order constraint.

12

The first part only adds (undirected or directed) edges to the graph. It could happen that the graph will have multiple edges between some pair of vertices v and w. The second part cleans up these multiple edges. It deletes all directed edges from v to w except the one with the largest gap-order constraint. By symmetry, it does the same for all directed edges from w to v.

By definition of a gap-graph there can be only one edge between v and w. If there are several edges (e.g., an undirected and a directed edge, or two edges with opposite directions), then the shortcut operation cannot return a gap-graph. Therefore it has to abort with an error message. If there is at most one edge between each pair of vertices, then what the shortcut operation will return is clearly a gap-graph. Note that the shortcut operation does not care whether this gap-graph is consistent.

An example of the shortcut operation is shown in Figure 3. The input gap-graph is shown in part (a) and the gap-graph obtained as a result of the shortcutting over vertex y is shown in part (b). There the shortcut creates three new directed edges, i.e., $x_1 <_6 x_2$, $x_1 <_1 x_3$, and $2 <_9 x_2$ and updates the gap value of one directed edge, i.e. edge $2 <_3 x_3$.

Definition 3.9 Let G_1 and G_2 be two gap-graphs over the same set of variables $v_1, \ldots, v_n$ and constants l and u, where $l, u \in D \cup \{+\infty, -\infty\}$ and $l < u$. Then a *merge* operation on G_1 and G_2 creates a gap-graph G with vertices $v_1, \ldots, v_n, v_{n+1} = l, v_{n+2} = u$ as follows. For each $0 < i, j \leq n + 2$ do the following.

If there is no edge between v_i and v_j in G_1 and G_2, then add no edge to G.
If there is an edge between v_i and v_j in only one of G_1 or G_2, then add that edge to G.
If $v_i = v_j$ in both G_1 and G_2, then add $v_i = v_j$ as an edge to G.
If $v_i <_{g_1} v_j$ in G_1 and $v_i <_{g_2} v_j$ in G_2, then add $v_i <_{max(g_1, g_2)} v_j$ as an edge to G.
If $v_i >_{g_1} v_j$ in G_1 and $v_i >_{g_2} v_j$ in G_2, then add $v_i >_{max(g_1, g_2)} v_j$ as an edge to G.
In any other case the merge operation fails, and it does not return any graph. $\square$

The intuition behind the merge operation is also simple. We want any assignment that satisfies the output gap-graph to satisfy both of the input gap-graphs. The operation guarantees this by checking that the corresponding edges in the two input gap-graphs are compatible and by adding always the edge which has the stricter gap-order constraint to the output gap-graph.

An example of the merge operation is shown in Figure 4. The input gap-graphs are Figure 3(a) and Figure 4(c) and the result of the merge operation is in part (d). There the merge operation creates a gap-graph such that on each edge the gap value is the maximum of the corresponding gap values in the two input gap-graphs. For example, $x_2 <_2 18$ in (a) and $x_2 <_3 18$ in (c) yields $x_2 <_3 18$ in (d).

Definition 3.10 Let G_1 and G_2 be two gap-graphs over the same set of variables $v_1, \ldots, v_n$ and constants $v_{n+1} = l$ and $v_{n+2} = u$, where $l, u \in D \cup \{+\infty, -\infty\}$ and $l < u$. We say that G_1 *subsumes* G_2 when $v_i <_{g_2} v_j$ (or $v_i = v_j$) is a directed (or undirected) edge in G_2 if and only if $v_i <_{g_1} v_j$ for some $g_1 \geq g_2$ (or $v_i = v_j$) is a directed (or undirected) edge in G_1. $\square$
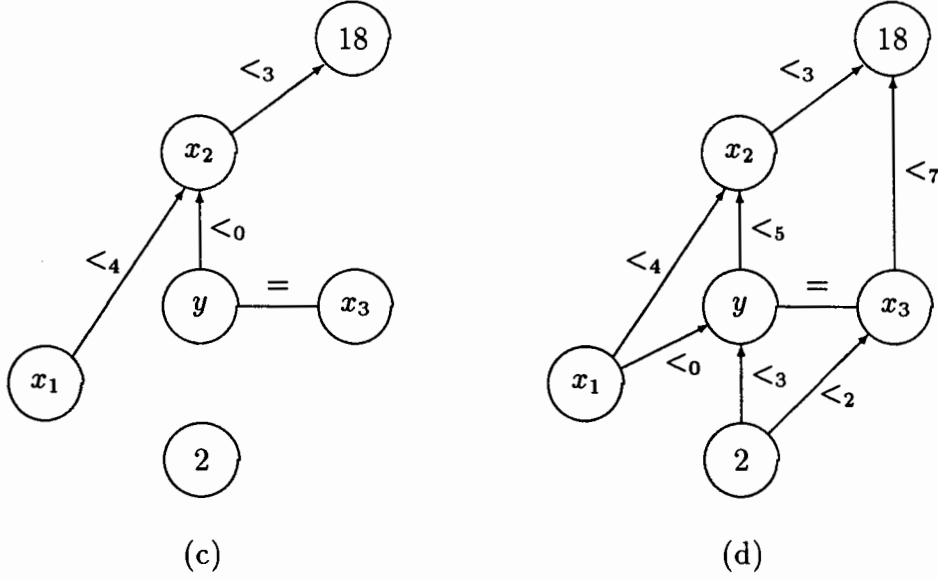
Figure 4: Example of merge of gap-graphs

The subsume operation serves to compare gap-graphs that have the same underlying graphs. For gap-graph A to subsume another gap-graph B, for each corresponding directed edge A must have a larger gap-order constraint than B has. The importance of this comparison will become clear in Section 3.3, where we show a visual representation of the notion of subsumption.

Next we give an example of subsume using Figure 5. The gap-graphs in Figure 3(a) and in Figure 5(e) have the same underlying graphs. By checking the conditions in Definition 3.10 we see that (e) subsumes (a).

The *extend* operation is the simplest of the four operations on gap-graphs. It adds an isolated vertex to the gap-graph. Since there are no edges incident on the isolated vertex, in itself extend is a useless operation. We use extend before a merge operation to augment the set of vertices to the same set in the gap-graphs to be merged.

After describing how the four operations are performed, we now show the semantic correctness of the operations defined, that is, we show that the operations are consistency preserving. This notion we make more precise for each case as we get to it. For the shortcut operation we want to show that the input gap-graph is consistent if and only if the output gap-graph is consistent.

Lemma 3.11 Let G be a gap-graph over variables $y, v_1, \ldots, v_n$ and constants $v_{n+1} = l$ and $v_{n+2} = u$ where $l, u \in D \cup \{+\infty, -\infty\}$ and $l < u$. Let G' be the gap-graph obtained by shortcutting over y in G (if exists). Let $a_0, a_1, \ldots, a_n$ be any sequence of integer numbers. Then $a_0, a_1, \ldots, a_n \models G$ if and only if G' exists and $a_1, \ldots, a_n \models G'$.

Proof: See Appendix. □

For the merge operation we want to show that the *and* of the input gap-graphs is consistent if and only if the output gap-graph is consistent.

14
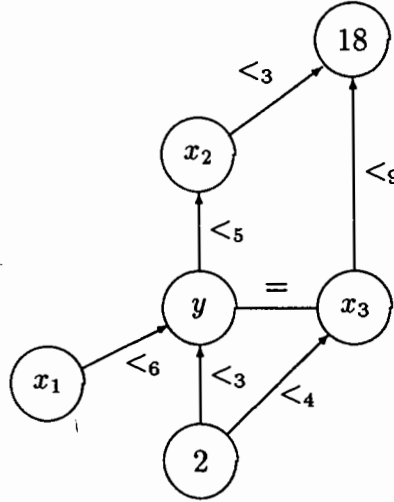
(e)

Figure 5: Example of subsume of gap-graphs

Lemma 3.12 Let G_1 and G_2 be two gap-graphs over the same set of variables $v_1, \ldots, v_n$ and constants $v_{n+1} = l$ and $v_{n+2} = u$, where $l, u \in D \cup \{+\infty, -\infty\}$ and $l < u$. Let G be the gap-graph obtained by merging G_1 and G_2 (if exists). Then for any assignment $\mathcal{A} = \{a_1, \ldots, a_n\}$, $\mathcal{A} \models G_1$ and $\mathcal{A} \models G_2$ if and only if G exists and $\mathcal{A} \models G$.

Proof: See Appendix. □

For the subsume operation we want to show that if a gap-graph G is consistent, then any gap-graph that G subsumes is consistent. Note that the reverse may not be true. Our evaluation method uses only the first direction.

Lemma 3.13 Let G_1 and G_2 be two gap-graphs over variables $v_1, \ldots, v_n$ and constants l and u, where $l, u \in D \cup \{+\infty, -\infty\}$ and $l < u$. If G_1 subsumes G_2, then for any assignment $\mathcal{A} = \{a_1, \ldots, a_n\}$ if $\mathcal{A} \models G_1$, then $\mathcal{A} \models G_2$.

Proof: By Definition 3.6, $\mathcal{A} \models G_1$ if and only if $\mathcal{A}$ satisfies all gap-order constraints on the edges. Assume that $\mathcal{A} \models G_1$ and G_1 subsumes G_2. It is easy to see that $\mathcal{A}$ also satisfies each gap-order constraint in G_2. Each gap-order constraint $a_i <_{g_2} a_j$ (or $a_i = a_j$) in G_2 holds because $a_i <_{g_1} a_j$ for some $g_1 \geq g_2$ (or $a_i = a_j$) is true in G_1 by Definition 3.10. Hence $\mathcal{A} \models G_2$. □

We also note that *extend* is obviously a consistency preserving operation because an isolated vertex can be assigned any integer value. The extended gap-graph is consistent if and only if the original gap-graph is consistent.

15

3.3 The Query Evaluation Method

In Section 3.3.1 we present our query evaluation algorithm called ALG for Datalog with integer order queries that are presented in *gap-graph form*. We also describe a simple procedure to find the gap-graph form of Datalog with integer order queries.

In Section 3.3.2 we show that ALG terminates on every Datalog with integer order query that is input in gap-graph form. To prove termination we transform the problem of query evaluation to a geometric problem, by representing each gap-graph as a point in some finite dimensional space. That enables the use of a geometric lemma (Lemma 3.14) to prove termination in Theorem 3.15.

In Section 3.3.3 we show that every Datalog with integer order query has a unique *full-model*, i.e., a unique fixpoint unrestricted relational database. We also show that what ALG returns is always some generalized output database (in gap-graph form) that is a finite description of the full-model.

3.3.1 The Query Evaluation Algorithm

At first we describe what we mean by *gap-graph form*. Let P be the program and d be the generalized input database of a Datalog with integer order query, where each generalized tuple is a conjunction of integer order constraints. By Lemma 3.3 we transform each generalized tuple of d into a semantically equivalent disjunction of gap-graphs.

Similarly, using Lemma 3.3 we transform each rule of P with a conjunction of integer order constraints into a semantically equivalent disjunction of rules with gap-graph constraints.

Clearly, in the algorithm within Lemma 3.3 we can choose l and u to be the smallest and largest constants in P. Using that choice, each gap-graph will have only l and u as constants in it. This we call the *gap-graph form* of Datalog with integer order queries.

Next we describe the query evaluation algorithm called ALG. The input of the query evaluation algorithm is a Datalog with integer order query P in gap-graph form. The output of the query evaluation algorithm is a generalized output database in gap-graph form. That is, for each IDB predicate of the form $R(x_1, \ldots, x_k)$ the algorithm produces a set of gap-graphs over vertices $x_1, \ldots, x_k, l, u$.

16

Query Evaluation Algorithm ALG

WHILE can add a gap-graph to the database **DO**

    BEGIN

        Add a gap-graph to the database using a *rule application* for some rule of the form $A_0(x_1, \ldots, x_m) :\!- A_1(\ldots), \ldots, A_k(\ldots), C$ in P, where $A_0, A_1, \ldots, A_k$ are relation symbols, C is a gap-graph, and the set of variables in the rule is $S = \{x_1, \ldots, x_m, y_1, \ldots, y_n\}$. A rule application consists of the following steps.

        (1) Pick for each A_i, $1 \leq i \leq k$ a gap-graph from the database.
        (2) Extend all gap-graphs to S.
        (3) Merge all gap-graphs.
        (4) Shortcut out vertices $y_1, \ldots, y_n$ to yield G.
        (5) Check that G is consistent and does not subsume another gap-graph for A_0 in the database. If true, add G for A_0 to the database.

END

    Let d be the output database. Suppose that G is a gap-graph over $x_1, \ldots, x_m, l, u$ that is derived for $A_0(x_1, \ldots, x_m)$ by one rule application during the query evaluation. Then G is a generalized tuple of the generalized output relation A_0 in d. We denote this as $A_0(G) \in d$.

3.3.2   A Termination Proof for ALG

To better analyze the termination of ALG, we provide first a visual interpretation of its operation. This visual interpretation maps each gap-graph in the database to a point in some fixed dimensional space. The spacial relations among the points will provide important clues to the relations among the gap-graphs. We do this as follows.

    Let G be a gap-graph with m directed edges. Let σ be an ordering of the directed edges in G. Let $B = (b_1, \ldots, b_m)$ be the m dimensional tuple obtained by listing the gap values in the labels of the directed edges in σ order. Then we can map G to a point in m-dimensional space.

    Suppose that $G_1, \ldots, G_n$ have the same underlying graph with m directed edges. We can map each G_i as a point B_i in m-dimensional space, using the same fixed ordering of directed edges. We know that G_i subsumes G_j if and only if each coordinate of B_i is $\geq$ the same coordinate of B_j. As with the gap-graphs, in this case we also say that B_i subsumes B_j.

    For example, looking at two gap-graphs A and B over x, l, u, where x is some variable and l and u are constants, assume that there are two directed edges in both A and B, namely $l <_4 x$ and $x <_5 u$ are edges in A and $l <_3 x$ and $x <_2 u$ are edges in B. Then we can map these gap-graphs to points in two dimension as shown in Figure 6. There is a visual representation for the fact that A subsumes B, and so does any point in the boxed region.
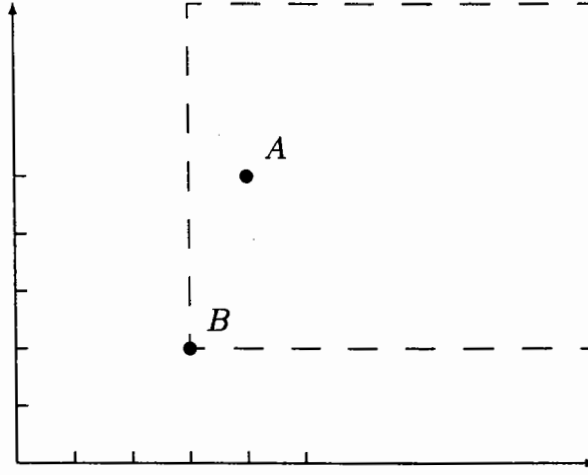
Figure 6: Geometric interpretation of subsumption

We will also use the following geometric lemma.

Lemma 3.14 In any fixed dimension, any sequence of distinct points with only natural number coordinates must be finite, if no point subsumes any earlier point in the sequence.

Proof: We prove the theorem by induction on the dimension k of the space in which the points lie. For $k = 0$, the whole space is only a single point, hence the theorem holds. Now we assume that the theorem holds for k dimensions and show that it is true for $k + 1$ dimensions.

Let S be any arbitrary sequence of points in which no point subsumes any earlier point. Let $x_1, \ldots, x_{k+1}$ be the coordinate axis of the $k+1$ dimensional space, and let $(a_1, \ldots, a_{k+1})$ be the first point in S. By the requirement of nonsubsuming and distinctness, for any later point $(b_1, \ldots, b_{k+1})$ for some i, $0 < i \leq k + 1$ it must be true that $b_i < a_i$. That means that any point in the sequence after $(a_1, \ldots, a_{k+1})$ must be within one or more of the k-dimensional regions $x_1 = 0$ or $x_1 = 1$ or $\ldots$, $x_1 = a_1$ or $\ldots x_{k+1} = 0$ or $x_{k+1} = 1$ or $\ldots x_{k+1} = a_{k+1}$.

As we add points to each of these regions from S, no point can subsume any earlier one within these regions. Therefore, by the induction hypothesis, only a finite number of points from S can be placed into each of these k-dimensional regions. Since the number of these regions is finite, S also must be finite. $\square$

Now we are ready to prove that ALG terminates on all of its inputs.

Theorem 3.15 Any Datalog with integer order query has a bottom-up evaluation with a gap-graph closed form.

Proof: Suppose that we use algorithm ALG to evaluate a query P. Let $R(x_1, \ldots, x_k)$ be any IDB predicate in P. All database facts for R are gap-graphs over $x_1, \ldots, x_k, l, u$.

Consider all the gap-graphs over these vertices. There is an infinite number of these gap-graphs. However, the gap-graphs can have only 4^{k+2} underlying graphs. That is because between each distinct pair of vertices v and w, we may have no edge, $v = w$, $v < w$, or $v > w$. For each of these underlying graphs, fix an ordering for the set of directed edges. Also, for each underlying graph with m directed edges create an m-dimensional picture. We have only a finite number of pictures and each picture has a finite dimension. After each rule application, if a gap-graph G is added to R, then map G to a point in the picture that corresponds to the underlying graph of G, using the fixed ordering. By Lemma 3.14 the mapping to each picture terminates. Since there are a finite number of IDB predicates, reasoning similarly to R for each we see that the bottom-up evaluation described in ALG terminates. $\square$

3.3.3 A Model for Datalog Queries with Integer Order

In the introduction we gave an unrestricted semantics to Datalog queries with integer order. According to the unrestricted semantics, a generalized database d_1 with a finite number of gap-graph tuples can be interpreted to be a finite description for some unrestricted relational database d_2 with a finite or infinite number of regular relational tuples. We call d_2 the *full version* of d_1. We make this more precise in the following definition.

Definition 3.16 Let d_1 be a generalized relational database with gap-graph tuples, and let d_2 be an unrestricted relational database. Then d_2 is the *full version* of d_1 if for all tuple t, $A(t) \in d_2$ if and only if exists G such that $t \models G$ and $A(G) \in d_1$. $\square$

The unrestricted semantics together with Tarski's fixpoint theorem can be used to show that Datalog with integer order queries have a unique unrestricted least model. However, the unrestricted least model using the *naive evaluation* implied by Tarski's fixpoint theorem would take an infinite number of iterations to evaluate. In this section we show that our query evaluation algorithm finds a finite description for the unrestricted least model. By Theorem 3.15 we know that this description is always some generalized database with gap-graph tuples, and it is evaluable in finite time.

Our query evaluation algorithm is nondeterministic, and it may return different generalized databases with gap-graph tuples. That is no cause for alarm. It simply means that there may be several finite descriptions for the same unrestricted least model.

Theorem 3.17 Let P be a Datalog with integer order program with generalized database d_1 in gap-graph form. Let d_2 be the full version of d_1. If $M_P(d_1)$ is *any* output of ALG on P and $L_P(d_2)$ is the output of the *naive evaluation* of $P[d_2/d_1]$ (with input d_2 instead of d_1) then $L_P(d_2)$ is the full version of $M_P(d_1)$.

Proof: (if) Let $M_P^i(d_1)$ denote the database after the ith rule application in ALG. We prove by induction on i that for all tuple t, if exists G such that $t \models G$ and $A(G) \in M_P^i(d_1)$, then $A(t) \in L_P(d_2)$. For $i = 0$ $M_P^0(d_1) = d_1$ and $d_2 \subseteq L_P(d_2)$, hence the claim holds. Now assume the claim for i and prove for $i + 1$.

Suppose the $i + 1^{th}$ rule application uses rule $A_0(x_1, \ldots, x_m) :\!- A_1(\ldots), A_2(\ldots),$ $\ldots, A_k(\ldots), C$ of P where C is a gap-graph and let $S = \{x_1, \ldots, x_m, y_1, \ldots, y_n\}$ be the variables in the rule. Suppose the application picks form $M_P^i(d_1)$ gap-graphs $G_1, \ldots, G_k$ for $A_1, \ldots, A_k$. Let $G_1', \ldots, G_k', C'$ be the extensions of the gap-graphs to S, M the merge of the extensions, and G the shortcut of M over $y_1, \ldots, y_n$. Assume that G is consistent and that $A_0(G)$ is added to $M_P^{i+1}(d_1)$.

Let $t = (a_0, \ldots, a_m)$ be any tuple such that $t \models G$. By Lemma 3.11 there exists a $B = \{a_0, \ldots, a_m, b_1, \ldots, b_n\}$ such that $B \models M$. By Lemma 3.12 $B \models G_1', \ldots, B \models G_k'$, and $B \models C'$. Hence, with the necessary restrictions $B \models G_1, \ldots, B \models G_k$, and $B \models C$. Note that for each $1 \leq j \leq k$, the jth restriction of B generates a tuple t_j such that $t_j \models G_j$. By the induction hypothesis, $t_j \models G_j$ and $A_j(G_j) \in M_P^i(d_1)$ implies that $A_j(t_j) \in L_P(d_2)$. Then by the naive evaluation $A_0(t) \in L_P(d_2)$ must be true. Reasoning similarly for each t, we see that for all t if $t \models G$ then $A_0(t) \in L_P(d_2)$. Since $A_0(G)$ is the only fact added in the $i + 1^{th}$ application, the claim holds.

(only if) Let $L_P^i(d_2)$ denote the database after the ith rule application in the naive evaluation. We prove by induction on i that for all tuple t, if $A(t) \in L_P^i(d_2)$, then there exists G such that $t \models G$ and $A(G) \in M_P(d_1)$. For $i = 0$ $L_P^0(d_2) = d_2$ and $d_1 \subseteq M_P(d_1)$, hence the claim holds. Now assume the claim for i and prove for $i + 1$.

Suppose the $i + 1^{th}$ rule application uses rule $A_0(x_1, \ldots, x_m) :\!- A_1(\ldots), A_2(\ldots),$ $\ldots, A_k(\ldots), C$ of P where C is a gap-graph and let $S = \{x_1, \ldots, x_m, y_1, \ldots, y_n\}$ be the variables in the rule. To perform the $i + 1^{th}$ rule application in the naive evaluation, assume that some assignment $B = \{a_1, \ldots, a_m, b_1, \ldots, b_n\}$ is chosen for the values of the variables in the rule. Each substitution of B into $A_1, \ldots, A_k$ yields a relational tuple $A_1(t_1), \ldots, A_k(t_k)$. Suppose that these are all present in $L_P^i(d_2)$ and that $B \models C$, and hence $A_0(t)$ where $t = (a_1, \ldots, a_m)$ is added to $L_P^{i+1}(d_2)$.

By the induction hypothesis there must be gap-graphs $G_1, \ldots, G_k$ such that $t_1 \models G_1, \ldots, t_k \models G_k$ and $A_1(G_1), \ldots, A_k(G_k) \in M_P(d_1)$. Hence with the necessary restrictions $B \models G_1, \ldots, B \models G_k$, and $B \models C$. Let M be the gap-graph obtained by merging the extensions of $G_1, \ldots, G_k$ and C. By Lemma 3.12 M exists and $B \models M$. Let G be the gap-graph obtained by shortcutting over vertices $y_1, \ldots, y_n$ in M. By Lemma 3.11 G exists and $B \models G$. Clearly G is consistent. If G does not subsume another gap-graph for A_0 in $M_P(d_1)$, then by the generalized evaluation $A_0(G) \in M_P(d_1)$ must be true. If G subsumes another gap-graph G' for A_0 in $M_P(d_1)$, then by Lemma 3.13 $B \models G'$. Hence either $t \models G$ and $A_0(G) \in M_P(d_d)$ or $t \models G'$ and $A_0(G') \in M_P(d_1)$ as required. By reasoning similarly for each t, we see that the claim holds. $\square$

Corollary 3.18 Let P be any generalized database logic program with generalized database d. Let $M_1(d)$ and $M_2(d)$ be the output of two different sequences of executions of ALG. Then for all tuple t, exists gap-graph G_1 such that $t \models G_1$ and $A(G_1) \in M_1(d)$ if and only if exists gap-graph G_2 such that $t \models G_2$ and $A(G_2) \in M_2(d)$. $\square$

We call each finite description output a *gap-model* of the Datalog with integer order query P. We also call the unrestricted least model the *full-model* $\mathcal{M}_P$ of P because it is

the set of relational tuples that satisfy any generalized relational tuple in any gap-model. Corollary 3.18 is another way of seeing that $\mathcal{M}_P$ is unique even though the gap-models are not.

4   The Recognition Problem

In this section we present a query evaluation algorithm called PALG that is designed to do efficiently tuple recognition. By tuple recognition we mean checking whether a given regular relational tuple is in $\mathcal{M}_P$, where $\mathcal{M}_P$ is the full-model of a given Datalog with integer order query (program with database) P. We saw in the previous section that $\mathcal{M}_P$ is a unique unrestricted (finite or infinite) set of regular relational tuples. Therefore the tuple recognition problem is well-defined. For a fixed Datalog with integer order program P and any generalized input database d, the complexity of performing the tuple recognition check in terms of the size of d we call the *generalized data complexity* of P (see also the definition in Section 2).

The tuple recognition problem is motivated by several reasons. For example, in the gap-model form the database output is hard to read, hence it is likely that users would ask for particular tuples instead of the whole generalized output database relation.

The query evaluation algorithm PALG is a modification of algorithm ALG of Section 3. The main idea is the following. Suppose that we are given a tuple t to be tested. Then let l to be the largest and u to be the smallest constant in t or the generalized database, which we assume is given in gap-graph form. Each gap-graph represents a set of integer tuples. Our observation is the following. Essentially each gap-graph is composed of three main parts (we make this precise later in Definition 4.1 and Lemma 4.2). First, those vertices which are below l, second, those vertices which are between l and u, and third, those vertices which are above u.

Our second observation is that the precise gap values are necessary to keep only for the vertices and edges in the second group, that is for vertices and edges that are between l and u. That is because the output gap-graph that we are interested in for the tuple recognition problem will be a special gap-graph, that is, it will have only vertices between l and u. For all the other gap-graphs in the generalized input database and output database instead of the precise gap values for edges and vertices in the first and third group, it is enough that we keep the directions only. We call these simplified gap-graphs the *partial graphs* (see Definition 4.3).

Algorithm PALG will always use as inputs and derive partial graphs. The desirable effect of this simplification is that we now have a well-defined upper bound on the size of the generalized output database that contains only partial graphs (see Definition 4.4 and Lemma 4.5).

The key reason the simplification works is that there is a limited interaction among the three main parts of gap-graphs. Note that in general gap-graphs may have other parts, for example, they may have isolated vertices that are indeterminate with respect to l and u.

Our technical definition of (l, u)-graphs limits the number of cases of possible interactions among the different parts of gap-graphs. As a result, we can show in Lemma 4.6 and Lemma 4.7 that a gap-graph G is derived by ALG if and only if the partial graph G' of G is derived by $PALG$.

For the tuple recognition problem it is enough to test whether t satisfies a gap-graph G in the full-model. Whenever t satisfies G, G has only vertices between l and u, and the partial graph of G is just itself. Hence testing whether t satisfies any of the partial graphs derived by PALG is enough. That is the intuition for Theorem 4.8.

Definition 4.1 Let G be a gap-graph over $x_1, \ldots, x_n, l, u$ with constants $l < u$. If for each vertex x_i in G, $x_i <_g l, x_i = l, x_i = u, x_i >_g u$ is an edge, or $l <_g x_i$ and $x_i <_h u$ are both edges in G for some g and h, then G is called an *(l,u)-graph.* $\square$

Lemma 4.2 Let G be a gap-graph over $x_1, \ldots, x_k, l, u$ with constants $l < u$. Let l', u' be any integers such that $l' \leq l$ and $u \leq u'$. Then G can be transformed into a disjunction of (l', u')-graphs over $x_1, \ldots, x_k, l', u'$ in $O(5^k k^2)$ time.

Proof: See Appendix. $\square$

Definition 4.3 Let G be an (l,u)-graph over $x_1, \ldots, x_n, l, u$ with constants $l < u$. We call the graph obtained by deleting all gap values in G except on the edges that are on a path from l to u in the compact graph of G the *partial graph* of G. $\square$

Definition 4.4 Let $A(x_1, \ldots, x_k)$ be a relational atom and G_p be the partial graph of some consistent (l,u)-graph over $x_1, \ldots, x_k, l, u$. Then $A(G_p)$ is called a *base*. $\square$

We call *p-application* the rule application with G replaced by the partial graph G_p of G and the subsumption check skipped in step (5). We call PALG the algorithm ALG with "gap-graph" replaced with "partial-graph" and "application" replaced with "p-application".

Lemma 4.5 Let P be a generalized database logic program with m IDB predicates each with arity $\leq k$. Let l be the smallest and u be the largest element of D. Then there are $m(u - l)^{O(k^2)}$ bases. PALG terminates after at most that many iterations.

Proof: Let $A(G_p)$ be any base. Note that by Definition 4.4 G_p is a consistent. Assume that A has arity k. For $k + 2$ vertices there are $(k + 2)(k + 1)/2$ distinct pairs. For each of these we may have no edge, an $=$ edge, a $<_g$ or a $>_g$ gap-order for some g. If the edge does not lie on a path from l to u in the compact graph of G_p, then the gap value must be absent (or 0). If it does, then by consistency and Lemma 3.6 the gap value $g < u - l$. Therefore for each $<_g$ and $>_g$ edge there can be at most $u - l$ different gap values. This gives at most $(2 + 2(u - l))^{(k+2)(k+1)/2}$ many choices for base $A(G_p)$. We can reason similarly for each m relational atom. Since PALG derives always bases, PALG terminates within that many iterations. $\square$

22

Lemma 4.6 Let $A_0 :\!- A_1, \ldots, A_k, C$ be a rule in a generalized database logic program P, and let $G_1, \ldots, G_k, C$ be (l,u)-graphs and $G'_1, \ldots, G'_k, C'$ be their partial graphs. Then a rule application derives $A_0(G)$ using $A_1(G_1), \ldots, A_k(G_k)$ and C if and only if a p-application derives $A_0(G')$ using $A_1(G'_1), \ldots, A_k(G'_k)$ and C', where G' is the partial-graph of G.

Proof: See Appendix. □

Lemma 4.7 Let P be a generalized database logic program with a database d_1 that has only (l,u)-graph constraints in the database. Let d_2 be the database which has exactly the partial form of each (l,u)-graph in d_1. Then algorithm ALG derives $A_0(G)$ using d_1 as input if and only if algorithm PALG derives $A_0(G')$ using d_2 as input, where G' is the partial-graph of G.

Proof: We can define generalized derivation trees. Structurally, the derivation trees for any $A_0(G)$ in d_1 is the same as for any $A_0(G')$ in d_2, assuming that ALG is run subsumption-test-free. Then the proof is by induction on the depth of the derivation trees using Lemma 4.6.

Suppose now that $A_0(G)$ is derived but not added to the database by ALG because there is already an $A_0(G'')$ such that G subsumes G''. By Lemma 3.10 there is no t such that $t \models G$ and $t \not\models G''$. By Theorem 3.17 P has the same full-model whether we add (just this time) G to the database or not. Hence, by induction on the derivation trees ALG finds the same full-model in a subsumption-test-free run as in a regular run. □

Theorem 4.8 Let t be any tuple. For any fixed generalized database logic program P with a gap-graph database we can test whether $A_0(t) \in \mathcal{M}_p$ in $O((u-l)^c + n^k)$ time, where c is some constant, n is the size of the database, k is the largest arity of a relation in P, and l is the smallest and u is the largest constant in either t or the database.

Proof: Transform each gap-graph to a disjunction of (l,u)-graphs. This takes $O(5^k k^2 n)$ time. Then make all (l,u)-graphs partial. This takes $O(k^2 n)$ time. Then run algorithm PALG. By Lemma 4.5 PALG will terminate after $(u-l)^{O(k^2)}$ iterations. For gap-graphs with k vertices, each merge, shortcut, and rule application takes $O(k^2)$ time. Let constant m be the largest number of relational atoms in any rule of P. To do one iteration we may need to substitute in from the database m bases. Therefore, to derive a new base needs $O((u-l)^{O(mk^2)})$ time. When PALG terminates test for each base $A(G')$ whether $t \models G'$. These tests require $O(k^2)$ for each base.

By Lemma 4.7 if $A(G')$ is a base in the database, then G' is the partial-graph of some G such that $A(G)$ would be in the database by running ALG subsumption-test-free instead of PALG. Since G' and G are (partial) (l,u)-graphs, for each vertex x of both G and G' one of $x <_g l$, $x = l$, $l <_g x <_h u$, $x = u$, or $x >_g u$ must be true for some g and h. Suppose that $t \models G'$. Since t satisfies all gap-order constraints in G' and all values within t are $\geq l$ or $\leq u$, the first and the last cases cannot happen. Hence every vertex

is equal to l or u or lies on a path from l to u in the compact graph of G'. Hence every edge will lie on a path from l to u, hence G' must be the same as G by Definition 4.3. Hence $t \models A(G)$ and by Theorem 3.17 $t \in \mathcal{M}_P$. $\square$

4.1 Piecewise Linear Programs

We can find another subset of queries for which the recognition test can be done efficiently, namely in NLOGSPACE data complexity. These are the piecewise linear Datalog programs of Ullman and Van Gelder [27].

Theorem 4.9 The recognition test for piecewise linear Datalog and generalized databases with gap-graph tuples can be done in NLOGSPACE.

Proof: As in Lemma 4.7 it is enough to consider only subsumption-test-free derivations of algorithm ALG. In piecewise linear recursive programs all derivations are chains with possible sideway branches of length one. That is, at every rule application only the previously derived IDB gap-graphs is used together with some EDB gap-graphs (the length one branches). Hence we only need to store a constant number of gap-graphs, each with a $O(\log u)$ size because there are only a constant, i.e., $O(k^2)$ where k is maximum arity of a relational atom, number of edges in any gap-graph, and the gap values are at most $u - l$ in size. Therefore we can derive in $NSPACE(\log n)$ any given relational tuple $A(t)$ that is derivable. As shown by Immerman [13], $NSPACE(\log n)$ is closed under complement. Hence, we can test in $NSPACE(\log n)$ whether $A(t)$ is derivable. $\square$.

5 Open Problems

There are a number of open problems. We list a few:
(1) Establish a good bound on the maximum gap size.
(2) Develop more and better pruning methods for rule applications.
(3) Find complexity class lower bounds. For example, can the lower bound be NP-hard?
(4) Is it possible to combine Relational Calculus with integer order and Datalog with integer order queries with the results in [7] and [15]?

References

[1] S. Abiteboul, V. Vianu. Procedural and Declarative Database Update Lan-

guages. *Proc. 7th ACM Symposium on Principles of Database Systems*, 240–250, 1988.

[2] A.K. Aylamazyan, M.M. Gilula, A.P. Stolboushkin, G.F. Schwartz. Reduction of the Relational Model with Infinite Domain to the Case of Finite Domains. *Proc. USSR Acad. of Science (Doklady)*, 286(2):308–311, 1986.

[3] E.F. Codd. A Relational Model for Large Shared Data Banks. *CACM*, 13(6):377–387, 1970.

[4] A. Colmerauer. *An Introduction to Prolog III. CACM*, 28(4):412–418, 1990.

[5] A.K. Chandra, D. Harel. Computable Queries for Relational Data Bases. *Journal of Computer and System Sciences*, 21:156–178, 1980.

[6] A.K. Chandra, D. Harel. Horn Clause Queries and Generalizations. *Journal of Logic Programming*, 2(1):1–15, 1985.

[7] J. Chomicki, T. Imielinski. Relational Specifications of Infinite Query Answers. *Proc. ACM SIGMOD International Conference on Management of Data*, 174–183, 1989.

[8] M. Dincbas, et al. The Constraint Logic Programming Language CHIP. *Proc. Fifth Generation Computer Systems*, Tokyo Japan, 1988.

[9] J. Ferrante, J.R. Geiser. An Efficient Decision Procedure for the Theory of Rational Order. *Theoretical Computer Science*, 4:227–233, 1977.

[10] J. Ferrante, C.W. Rackoff. *The Computational Complexity of Logical Theories*, Springer-Verlag (No. 718), 1979.

[11] R. Hull, J. Su. Domain Independence and the Relational Calculus. *Technical Report* 88–64, University of Southern California, submitted to IPL, 1989.

[12] N. Immerman. Relational Queries Computable in Polynomial Time. *Information and Control*, 68:86-104, 1986.

[13] N. Immerman. Nondeterministic Space is Closed under Complement. SIAM Journal on Computing, 17:935-938, 1988.

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

[15] F. Kabanza, J-M. Stevenne, P. Wolper. Handling Infinite Temporal Data, *9th ACM Symposium on Principles of Database Systems*, 392-403, 1990.

[16] P.C. Kanellakis. Elements of Relational Database Theory. *Handbook of Theoretical Computer Science*, Vol. B, chapter 17, (J. van Leeuwen, A.R. Meyer, N. Nivat, M.S. Paterson, D. Perrin editors), North-Holland, 1990.

[17] P.C. Kanellakis, G.M. Kuper, P.Z. Revesz. Constraint Query Languages, *9th ACM Symposium on Principles of Database Systems*, 299-313, 1990.

[18] M. Kifer. On Safety, Domain Independence, and Capturability of Database Queries. *Proc. International Conference on Databases and Knowledge Bases*, Jerusalem Israel, 1988.

[19] S.C. Kleene. General Recursive Functions on Natural Numbers. *Matematische Annalen*, 112:727-742, 1936.

[20] P. Kolaitis, C.H. Papadimitriou. Why not Negation by Fixpoint? *Proc. 7th ACM Symposium on Principles of Database Systems*, 231–239, 1988.

[21] H.R. Lewis, C.H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 1981.

[22] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1984.

[23] Y.N. Moschovakis. *Elementary Induction on Abstract Structures*. North Holland, 1974.

[24] R. Ramakrishnan. Magic Templates: A Spellbinding Approach to Logic Programs. *Proc. 5th International Conference on Logic Programming*, 141–159, 1988.

[25] P.Z. Revesz. A Closed Form for Datalog Queries with Integer Order. *Proc. 3rd International Conference on Database Theory*, 187–201, 1990.

[26] J.D. Ullman. *Principles of Database Systems*. Computer Science Press, 2nd Ed., 1982.

[27] J.D. Ullman, A. Van Gelder. Parallel Complexity of Logical Query Programs. *Algorithmica*, 3:5-42, 1988.

[28] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[29] M.Y. Vardi. The Complexity of Relational Query Languages. *Proc. 14th ACM Symposium on Theory of Computing*, 137–146, 1982.

A Appendix

Proof of Lemma 3.3: Let C be any conjunction of integer order constraints. Let $x_1, \ldots, x_n$ be the variables, l be the smallest and u be the largest constant in C. If there are no constants in C, let $l = -\infty$ and $u = +\infty$. If there is only one constant c in C, let $l = c$ and $u = +\infty$. First replace in C all constraints of the form $v \leq w$ by $((v < w) \vee (v = w))$, of the form $v \neq w$ by $((v < w) \vee (v > w))$, and of the form $v = c$ by $((c - 1 < v) \wedge (v < c + 1))$, where v and w are either variables or constants and c is a constant with $l < c < u$. Next put C into disjunctive normal form (i.e., disjunction of conjunctions). For each disjunct (i.e., conjunction of constraints) of C we create a gap-graph on $x_1, \ldots, x_n, l, u$ as follows.

Let C' be any disjunct. We check first whether there are any constants c_1 and c_2 with $c_1 < c_2$ such that $c_1 < c_1$, $c_1 > c_1$, $c_1 = c_2$, $c_1 > c_2$, or $c_2 < c_1$ occurs as a constraint in C', or whether there are any variables or constants v and w such that more than one of $v = w$, $v < w$ (or $w > v$), and $v > w$ (or $w < v$) occur as constraints in C'. If there are, then C' is clearly unsatisfiable, hence we need not create a gap-graph G for it. Otherwise, take each constraint in C' in order.

If the constraint is $x_i = x_j$, then add $x_i = x_j$ as edge to G.
If the constraint is $x_i < x_j$ (or $x_j > x_i$), then add $x_i <_0 x_j$ as edge to G.
If the constraint is $x_i = l$ (or $x_i = u$), then add $x_i = l$ (or $x_i = u$) as edge to G.
If the constraint is $c < x_i$ for some constant $l \leq c \leq u$ and this is the largest lower bound on x_i within C', then add $l <_{(c-l)} x_i$ as edge to G.
If the constraint is $x_i < c$ for some constant $l \leq c \leq u$ and this is the smallest upper bound on x_i within C', then add $x_i <_{(u-c)} u$ as edge to G.

It is easy to see that this algorithm yields a gap-graph for each C', such that the conjunction of the gap-order constraints within the gap-graph is equivalent to C'. □

Proof of Lemma 3.7: In this proof we use G_c to denote the compact graph of G and the function $llp(v, w)$ for vertices v and w to denote the length of the longest path from v to w in G_c. We measure all path lengths in G_c. At first assume that $l, u \in D$ and that there is no path from u to l.

(if) Assume that the gap-graph G is acyclic and the longest path from l to u is less than $u - l$. Let m be the length of the longest path in G_c. Then we will make a consistent assignment to $v_1, \ldots, v_n$ as follows.

For each vertex v with a path from l or u to v in G_c, assign the maximum of values $l + llp(l, v) + 1$ and $u + llp(u, v) + 1$. For each leaf vertex (other than possibly l or u) assign the value $(l - m - 1)$. For each non-leaf vertex w, if there is no path from l or u to w, assign the maximum of the values $(l - m - 1) + max_x llp(x, w) + 1$, where x is any leaf node in the gap-graph.

This assignment satisfies all the gap-order constraints in G. The only potential difficulty is in showing that all upper bounds are satisfied involving u during the first set of assignments and involving all first-group vertices during the second set of assignments.

It is easy to see that the first group of vertices are assigned as low values as possible. Let v be any vertex in the first group. If $v <_g u$ is an edge in G, then $value(v) = l + llp(l, v) + 1$ by our assignment and by acyclicity. Also, since $llp(l, u) < u - l$, $value(v) + g = l + llp(l, v) + 1 + g \leq l + llp(l, u) < l + (u - l) = u$. Hence, $value(v) + g < u$ as required.

Similarly, the second group of non-leaf vertices are also assigned as low values as possible. Let w be any vertex in the second group. If $w <_g v$ is an edge in G, with w in the second group and v in the first group, then by our assignment $value(w) = (l - m - 1) + max_x llp(x, w) + 1$ where x is any leaf node. Since for each leaf node x $llp(x, v) \leq m$, $value(w) + g = (l - m - 1) + max_x llp(x, w) + 1 + g \leq (l - m - 1) + m = l - 1 < value(v)$. Hence $value(w) + g < value(v)$ as required.

(only if) If G_c has a cycle, the length of the longest path from l to u is $\geq (u - l)$, or there is path from u to l then clearly there is no consistent assignment.

When $l = -\infty$ or $u = +\infty$ and there is a directed edge $(v, l) = (v, -\infty)$ or $(u, v) = (+\infty, v)$, then clearly there is no consistent assignment. Otherwise, we simply treat G as having no l or u in it, and do the above proof with the necessary simplifications. $\square$

Proof of Lemma 3.11: (if) Assume that G' exists and assignment $\mathcal{A}' = \{a_1, \ldots, a_n\} \models G'$. Within G let $u_1 <_{g_1} y, u_2 <_{g_2} y, \ldots, u_m <_{g_m} y$ be the edges incident on y with a gap-order label that constraints y from below, where the u's are vertices. Also within G let $y <_{f_1} w_1, y <_{f_2} w_2, \ldots, y <_{f_n} w_n$ be the edges incident on y with a gap-order label that constraints y from above, where the w's are vertices.

We want to find an assignment $\mathcal{A} = \{a_0, \ldots, a_n\}$ for the vertices in G that satisfies all the gap-order constraints in the edges. Take assignment $\mathcal{A}'$ as a substitution for all the vertices save for y. Then find k for which $u_k + g_k$ is maximum and p for which $w_p - f_p$ is minimum.

To obtain G' form G by a shortcut operation over y we added in case 4 a constraint between each u_i and w_j, $1 \leq i \leq m$ and $1 \leq j \leq n$. In particular, we added the constraint $u_k <_{g_k + f_p + 1} w_p$. Clearly, it is sufficient and necessary to choose a value a_0 for y such that $u_k + g_k < a_0 < w_p - f_p$ to satisfy all lower and upper bounds on y within G.

Since $\mathcal{A}' \models G'$, there are at least $g_k + f_p + 1$ integers between u_k and w_p after the assignment. Therefore, it is always possible to pick an integer a_0 such that there are at least g_k integers between u_k and a_0 and at least f_p integers between a_0 and w_p. If there is no v_i such that $y = v_i$ is a constraint in G, then pick any such a_0. Then $\mathcal{A} \models G$.

If $y = v_i$ for some v_i in G, then let $a_0 = a_i$. To obtain G' from G by a shortcut operation over y we added in cases 2 and 3 as constraints $u_1 <_{g_1} v_i, u_2 <_{g_2} v_i, \ldots, u_m <_{g_m} v_i$ and $v_i <_{f_1} w_1, v_i <_{f_2} w_2, \ldots, v_i <_{f_n} w_n$. Therefore, a_0 satisfies all the lower and upper bounds on y. If any other $v_j = y$, then by case 1 of the shortcut we know that $v_i = v_j$ is a constraint in G', hence $a_i = a_j$. Therefore, our choice for a_0 satisfies all constraints on y. Hence $\mathcal{A} \models G$.

(only if) Let $\mathcal{A} = \{a_0, a_1, \ldots, a_n\}$ be an integer assignment to the variables of G, such that $\mathcal{A} \models G$. Clearly, for all $0 < i, j, \leq n + 2$, if $v_i = y$ and $y = v_j$, then $v_i = v_j$ holds, if $v_i = y$ and $y <_g v_j$, then $v_i <_g v_j$ holds, if $v_i <_g y$ and $y = v_j$, then $v_i <_g v_j$ holds, and if $v_i <_{g_i} y$ and $y <_{g_j} v_j$, then $v_i <_{g_i + g_j + 1} v_j$ holds. Note that the shortcut operation adds only these types of constraints in the first step. Therefore, $\mathcal{A}$ is a consistent assignment to the graph after the first step. The deletions in the second step leave between each v_i and v_j possibly one gap-order $v_i = v_j$ (if there was one), possibly one gap-order $v_i <_{g_1} v_j$ (if there was any gap-order of the form $v_i <_{g_2} v_j$), and possibly one gap-order $v_i >_{h_1} v_j$ (if there was any gap-order of the form $v_i >_{h_2} v_j$). If there were more than one edge between any v_i and v_j, then the graph would be inconsistent and the shortcut operation would fail. However, deleting constraints must leave the graph consistent, hence the shortcut operation cannot fail and it will return a gap-graph G'. Therefore, $\mathcal{A}' = \{a_1, \ldots, a_n\}$ is an integer assignment to the variables in G' such that $\mathcal{A}' \models G'$. $\square$

Proof of Lemma 3.12: (if) Assume that G exists and $\mathcal{A} \models G$. By Definition 3.6, $\mathcal{A}$ satisfies all gap-order constraints on the edges in G. We show that $\mathcal{A}$ also satisfies all gap-order constraints on the edges in G_1 and G_2.

Let v_i and v_j be any pair of vertices. If $v_i = v_j$ in G_1, then by either case 2 or 3 the merge operation adds $v_i = v_j$ to G. If $v_i <_g v_j$ in G_1, then by either case 2 or 4 the merge operation adds $v_i <_h v_j$ to G for some $h \geq g$. If $v_i >_g v_j$ in G_1, then by either case 2 or 5 the merge operation adds $v_i >_h v_j$ to G for some $h \geq g$. Hence, if there is a gap-order constraint between any pair of vertices in G_1, then there is an equal or stronger constraint between that pair of vertices in G. Therefore, $\mathcal{A} \models G_1$ must be true. By symmetry $\mathcal{A} \models G_2$ must also be true.

(only if) Assume that $\mathcal{A} \models G_1$ and $\mathcal{A} \models G_2$. Clearly, if $v_i = v_j$ or $v_i <_g v_j$ or $v_i >_h v_j$ in either G_1 or G_2, then $v_i = v_j$ or $v_i <_g v_j$ or $v_i >_h v_j$ holds, if $v_i = v_j$ in G_1 and $v_i = v_j$ in G_2, then $v_i = v_j$ holds, if $v_i <_{g_1} v_j$ in G_1 and $v_i <_{g_2} v_j$ in G_2, then $v_i <_{max(g_1,g_2)} v_j$ holds, if $v_i >_{g_1} v_j$ in G_1 and $v_i >_{g_2} v_j$ in G_2, then $v_i >_{max(g_1,g_2)} v_j$ holds. Note that cases 2 to 4 of the merge operation add only these types of constraints to G. No $\mathcal{A}$ can satisfy more than one of the gap-order constraints $v_i = v_j$, $v_i <_g v_j$, and $v_i >_h v_j$ for any $g, h \geq 0$. Therefore, the last case in Definition 3.9 cannot occur and the merge cannot fail. Therefore G exists and $\mathcal{A} \models G$ must be true. $\square$

Proof of Lemma 4.2: Create two new vertices labeled l' and u'. For each x_i select one of the constraints $x_i < l'$, $x_i = l'$, $x_i = u'$, $x_i > u'$, or $l' < x_i$ and $x_i < u'$, and add that to G, unless there are equal or stronger gap-order constraints already present in G. After the selected constraints are added by Definition 4.1 we obtain an (l,u)-graph. We can make 5^k selections. For these (l,u)-graphs the set of consistent assignments is disjoint.

We can transform an (l,u)-graph into an (l',u')-graph as follows. For each edge $x_i <_g l$ add $x_i <_{g+u'-l} u'$, for each $x_i = l$ add $l' <_{l-l'-1} x_i$ and $x_i <_{u'-l-1} u'$, for each $l <_g x_i$ add $l' <_{g+l-l'} x_i$, for each $x_i <_g u$ add $x_i <_{g+u'-u} u'$, for each $x_i = u$ add $l' <_{u-l'-1} x_i$ and $x_i <_{u'-u-1} u'$, and for each $u <_g x_i$ add $l' <_{g+u-l'} x_i$. Finally, delete the vertices labeled l and u and all edges incident on them. For each (l,u) graph this takes $O(k)$ time. $\square$

Proof of Lemma 4.6: Suppose that $A_0(G)$ is derived by a rule application. Then G is consistent by the check in step (5) of the rule application. Let M be the merge of the extensions of $G_1, \ldots, G_k$, and C. By Lemma 3.11 M is consistent and by Lemma 3.12 $G_1, \ldots, G_k$ and C are also consistent. By Lemma 3.6 they are all acyclic.

Since $G_1, \ldots, G_k$ and C are (l,u)-graphs and every v_i appears in at least one of them, $v_i <_g l$, $v_i = l$, $v_i = u$, $v_i >_g u$, or $l <_g v_i$ and $v_i <_h u$ must be a constraint in at least one of them, for some g and h. Therefore, M will also have one of those constraints for each v_i. Hence, M will be an (l,u)-graph. Moreover, whenever v_i appears in any G_j, then v_i must have the same constraint on it as in M, ignoring gap values, because the five cases are completely disjoint.

Let $v_i <_g v_j$ for some g be an edge on a path from l to u in the compact graph of M. By acyclicity, $v_i <_g l$, $v_i = u$, and $u <_g v_i$ cannot be constraints in M for any g. Since M is an (l,u)-graph, either $l = v_i$ or $l <_g v_i <_h u$ must hold for some g and h. Similarly, either $l <_g v_j <_h u$ or $v_j = u$ must hold for some g and h. Hence $(l = v_i$ or $l <_g v_i <_h u)$

29

and ($l <_{g'} v_j <_{h'} u$ or $v_j = u$) must be in each G_j that contains v_i and v_j. Therefore, if G_j has a directed edge (v_i, v_j), then it lies on a path from l to u in the compact graph of G_j. By reasoning similarly for each edge on a path from l to u, we see that they all preserve their gap values in G'_j by Definition 4.3. Every other edge in G'_j looses its gap-value.

The merge operation takes the maximum of the gap values on each edge. Clearly, M and the merge M' have the same set of edges, ignoring gap values. The merge operation takes the maximum of the gap values of each edge (v_i, v_j) form each G_j and C for M and for each G'_j and C' for M'. Therefore, for each edge on a path from l to u in M the gap value will be the same as for the same edge in M', while all other edges in M' will have no gap value. Therefore M' is the partial-graph of M.

The shortcut operation over a vertex y can only shorten paths between any two other vertices, and it can neither create nor destroy a path between them. Hence any edge (v_i, v_j) that lies on a path from l to u in G must lie on a path from l to u in M or there must be edges (v_i, y) or $v_i = y$ and (y, v_j) or $v_j = y$ in M that lie on a path from l to u. Then the same edges with the same gap values are also in M'. Hence (v_i, v_j) with the same gap value will be in the shortcut G'' of M' as in G. Therefore, the longest path from l to u is less than $(u - l)$ in G if and only if it is true in G'' if and only if it is true in G' the partial-graph of G''. By Lemma 3.6 G is consistent if and only if G'' is consistent if and only if G' is consistent. Hence, $A_0(G)$ is derived by a rule application if and only if $A_0(G')$ is derived by a p-application. $\square$