Constraint-Generating Dependencies *

Marianne Baudinet
Université Libre de Bruxelles

Jan Chomicki
Monmouth University†

Pierre Wolper
Université de Liège‡

April 16, 1999

Abstract

Traditionally, dependency theory has been developed for uninterpreted data. Specifically, the only assumption that is made about the data domains is that data values can be compared for equality. However, data is often interpreted and there can be advantages in considering it as such, for instance obtaining more compact representations as done in constraint databases. This paper considers dependency theory in the context of interpreted data. Specifically, it studies *constraint-generating dependencies*. These are a generalization of equality-generating dependencies where equality requirements are replaced by constraints on an interpreted domain. The main technical results in the paper are a general decision procedure for the implication and consistency problems for constraint-generating dependencies, and complexity results for specific classes of such dependencies over given domains. The decision procedure proceeds by reducing the dependency problem to a decision problem for the constraint theory of interest, and is applicable as soon as the underlying constraint theory is decidable. The complexity results are, in some

*This work was supported by NATO Collaborative Research Grant CRG 940110 and by NSF Grant IRI-9110581.

†Dept. of Computer Science, Howard Hall, West Long Branch, NJ 07764, USA, Email: `chomicki@moncol.monmouth.edu`.

‡Institut Montefiore, B28, 4000 Liège Sart-Tilman, Belgium, Email: `pw@montefiore.ulg.ac.be`.

cases, directly lifted from the constraint theory; in other cases, optimal complexity bounds are obtained by taking into account the specific form of the constraint decision problem obtained by reducing the dependency implication problem.

1 Introduction

Relational database theory is largely built upon the assumption of uninterpreted data. While this has advantages, mostly generality, it foregoes the possibility of exploiting the structure of specific data domains. The introduction of constraint databases [25] was a break with this uninterpreted-data trend. Rather than defining the extension of relations by an explicit enumeration of tuples, a constraint database uses constraint expressions to implicitly specify sets of tuples. Of course, for this to be possible in a meaningful way, one needs to consider interpreted data, that is, data from a specific domain on which a basic set of predicates and functions is defined. A typical example of constraint expressions and domain are linear inequalities interpreted on the reals. The potential gains from this approach are in the compactness of the representation (a single constraint expression can represent many, even an infinite number of, explicit tuples) and in the efficiency of query evaluation (computing with constraint expressions amounts to manipulating many tuples simultaneously).

Related developments have concurrently been taking place in temporal databases. Indeed, time values are intrinsically interpreted and this can be exploited for finitely representing potentially infinite temporal extensions. For instance, in [24] infinite temporal extensions are represented with the help of periodicity and inequality constraints, whereas in [11, 12] and [2] deductive rules over the integers are used for the same purpose. Constraints have also been used recently for representing incomplete temporal information [28, 29, 43].

If one surveys the existing work on databases with interpreted data and implicit representations, one finds contributions on the expressiveness of the various representation formalisms [5, 4, 3, 18, 19, 7, 37], on the complexity of query evaluation [33, 43, 13], and on data structures and algorithms to be used in the representation of constraint expressions and in query evaluation [36, 8, 26, 40, 9, 41]. However, much less has been done on extending other parts of traditional database theory, for instance schema design and dependency theory. It should be clear that dependency theory is of interest in this context. For instance, in [23], one finds a taxonomy of dependencies that are useful for temporal databases. Moreover, many *integrity constraints* over interpreted data can be represented as generalized dependencies. For instance, the integrity constraints over databases with ordered domains studied in [22, 44] can be represented as generalized dependencies. Also, some versions of the constraint checking problem studied in [21] can be viewed as generalized dependency implication problems.

One might think that the study of dependency theory has been close to exhaustive. While this is largely so for dependencies over uninterpreted data (that is, the context in which data values can only be compared for equality) [39], the situation is quite different for dependencies over data domains with a richer structure. The subject of this paper is the theory of these interpreted dependencies.

Specifically, we study the class of *constraint-generating dependencies*. These are the generalization of equality-generating dependencies [6], allowing arbitrary constraints on the data domain to appear wherever the latter only allow equalities. For instance, a constraint-generating dependency over an ordered domain can specify that if the value of an attribute A in a tuple t_1 is less than the value of the same attribute in a tuple t_2, then an identical relation holds for the values of an attribute B. This type of dependency can express a wide variety of constraints on the data. For instance, most of the temporal dependencies appearing in the taxonomy of [23] are constraint-generating dependencies.

Our technical contributions address the implication and the consistency[1] problems for constraint-generating dependencies. The natural approach to these problems is to write the dependencies as logical formulas. Unfortunately, the resulting formulas are not just formulas in the theory of the data domain. Indeed, these formulas also contain uninterpreted predicate symbols representing the relations and thus are not a priori decidable, even if the data domain theory is decidable.

To obtain decision procedures, we show that the predicate symbols can be eliminated. Since the predicate symbols are implicitly universally quantified, this can be viewed as a form of second-order quantifier elimination. It is based on the fact that it is sufficient to consider relations with a small finite number of tuples. This then allows quantifier elimination by explicit representation of the possible tuples. The fact that one only needs to consider a small finite number of tuples is analogous to the fact that the implication problem for functional dependencies can be decided over 2-tuple relations [32]. Furthermore, for pure functional dependencies, our quantifier elimination procedures yields

[1]Though consistency is always satisfied for equality-generating dependencies, more general constraints turn it into a nontrivial problem.

exactly the usual reduction to propositional logic. For more general constraint dependencies, it yields a formula in the theory of the data domain. Thus, if this theory is decidable, the implication and the consistency problems for constraint-dependencies are also decidable. Our approach is based on simple general logical arguments and provides a clear and straightforward justification for the type of procedure based on containment mappings used for instance in [21].

The complexity of the decision procedure depends on the specific data domain being considered and on the exact form of the constraint dependencies. We consider three typical constraint languages: equalities/inequalities, ordering constraints, and linear arithmetic constraints. We give a detailed picture of the complexity of the implication problem for dependencies over these theories and show the impact of the form of the dependencies on tractability.

2 Constraint-Generating Dependencies

Consider a relational database where some attributes take their values in specific domains, such as the integers or the reals, on which a set of predicates and functions are defined. We call such attributes *interpreted*. The domain of an interpreted attribute, together with the functions and predicates defined on that domain constitute a structure to which corresponds a first-order language. Since it is usual to refer to the predicates we will be dealing with as "constraints", we will refer to the first-order language of an interpreted attribute's domain as a *constraint language* or *constraint theory* consisting of *constraint formulas* or *constraints*. For the simplicity of the presentation, let us assume that the database only contains one (universal) relation r and let us ignore the noninterpreted attributes. In this context, it is natural to generalize the notion of equality-generating dependency [6]. Rather than specifying the propagation of equality constraints, we write similar statements involving arbitrary constraints (i.e., arbitrary formulas in the theory of the data domain). Specifically, we define *constraint-generating k-dependencies* as follows (the constant k specifies the number of tuples the dependency refers to).

Definition 2.1 Given a relation r, a *constraint-generating k-dependency* over r (with $k \geq 1$) is a first-order formula of the form

$$(\forall t_1) \cdots (\forall t_k) \Big[\Big[r(t_1) \wedge \cdots \wedge r(t_k) \wedge C[t_1, \ldots, t_k] \Big] \Rightarrow C'[t_1, \ldots, t_k] \Big]$$

where $C[t_1, \ldots, t_k]$ and $C'[t_1, \ldots, t_k]$ denote arbitrary constraint formulas relating the values of various attributes in the tuples $t_1, \ldots, t_k$. There are no restrictions on these formulas, they can include all constructs of the constraint theory under consideration, including constants and quantification on the constraint domain. For instance, a constraint $C[t_1, t_2]$ could be $\exists z (t_1[A] < z \wedge z < t_2[A] < a)$.

Note that we have defined constraint-generating dependencies in the context of a single relation, but the generalization to several relations is immediate.

Constraint-generating 1-dependencies as well as constraint-generating 2-dependencies are the most common. Notice that functional dependencies are a special form of constraint-generating 2-dependencies. Constraint-generating dependencies can naturally express a variety of arithmetic integrity constraints. The following examples illustrate their definition and show some of their potential applications.

Example 2.1 In [23], an exhaustive taxonomy of dependencies that can be imposed on a temporal relation is given. Of the more than 30 types of dependencies that are defined there, all but 4 can be written as constraint-generating dependencies. These last 4 require a generalization of tuple-generating dependencies [6] (see Section 5).

In temporal databases, two basic temporal dimensions have been identified: *valid time* (the time when an event happened in the real world) and *transaction time* (the time when an event was recorded in the database). Thus, consider a relation $r(tt, vt)$ with two temporal attributes: valid time (vt) and transaction time (tt). The property "an event can only be recorded when it happens or within c time instants afterwards" is called "r being strongly retroactively bounded with bound $c \geq 0$" [23]. This property is expressed as the constraint-generating 1-dependency

$$(\forall t_1)\Big[r(t_1) \Rightarrow [(t_1[tt] \leq t_1[vt] + c) \wedge (t_1[vt] \leq t_1[tt])]\Big].$$

Another property, "there are no updates to the past," is called "r being globally nondecreasing" [23]. It is expressed as the constraint generating 2-dependency

$$(\forall t_1)(\forall t_2)\Big[[r(t_1) \wedge r(t_2) \wedge (t_1[tt] < t_2[tt])] \Rightarrow (t_1[vt] \leq t_2[vt])\Big].$$

∎

Example 2.2 Let us consider a relation $emp(name, boss, salary)$. Then the fact that an employee cannot make more than her boss is expressed as

$$(\forall t_1)(\forall t_2)\Big[[emp(t_1) \wedge emp(t_2) \wedge (t_1[boss] = t_2[name])] \Rightarrow (t_1[salary] \leq t_2[salary])\Big].$$

∎

3 Decision Problems for Constraint-Generating Dependencies

There are two basic decision problems for constraint-generating dependencies.

- *Implication*: Does a finite set of dependencies D imply a dependency d_0?
- *Consistency*: Does a finite set of dependencies D have a non-trivial model, that is, is D true in a nonempty relation?

The implication problem is a classical problem of database theory. Its practical motivation comes from the need to detect redundant dependencies, that is, those that are implied by a given set of dependencies. It is also the basis for proving the equivalence of dependency sets, and consequently for finding covers with desirable properties, such as minimality. The consistency problem has a trivial answer for uninterpreted dependencies: every set of equality- and tuple-generating dependencies has a 1-element model. However, even a single constraint-generating dependency may be inconsistent, as illustrated by

$$(\forall t)[r(t) \Rightarrow t[1] < t[1]].$$

We only study the *implication* problem since the consistency problem is its dual: a set of dependencies D is inconsistent if and only if D implies a dependency of the form:

$$(\forall t)[r(t) \Rightarrow C]$$

where C is any unsatisfiable constraint (we assume the existence of at least one such unsatisfiable constraint formula).

The result we prove in this section is that the implication problem for constraint-generating dependencies reduces to the validity problem for a formula in the underlying constraint theory. Specific dependencies and theories will be considered in Section 4, and the corresponding complexity results provided. The reduction proceeds in three steps. First, we prove that the implication problem is equivalent to the implication problem restricted to finite relations of bounded size. Second, we eliminate from the implication to be decided the second-order quantification (over relations). Third, we eliminate the first-order quantification (over tuples) from the dependencies themselves and replace it by quantification over the domain – a process that we call *symmetrization*. This gives us the desired result.

3.1 Statement of the Problem and Notation

Let r denote a relation with n interpreted attributes. Let $d_0, d_1, \ldots, d_m$ denote constraint-generating k-dependencies over the attributes of r. The value of k does not need to be the same for all d_i's. We denote by k_0 the value of k for d_0.

The *dependency implication problem* consists in deciding whether d_0 is implied by the set of dependencies $D = \{d_1, \ldots, d_m\}$. In other words, it consists in deciding whether d_0 is satisfied by every interpretation that satisfies D, which can be formulated as

$$(\forall r)\Big[r \models D \Rightarrow r \models d_0\Big], \tag{1}$$

where D stands for $d_1 \wedge \cdots \wedge d_m$. We equivalently write (1) as

$$(\forall r)\Big[D(r) \Rightarrow d_0(r)\Big]$$

when we wish to emphasize the fact that the dependencies apply to the tuples of r.

3.2 Towards a Decision Procedure

3.2.1 Reduction to k-tuple Relations

The following three lemmas establish that when dealing with constraint-generating k-dependencies, it is sufficient to consider relations of size[2] k. Their proofs are straightforward.

[2]In what follows, we consider relations as multisets rather than sets. This has no impact on the implication problem, but simplifies our procedure, starting with Lemma 3.1.

Lemma 3.1 *Let d denote any constraint-generating k-dependency. If a relation r does not satisfy d, then there is a relation r' of size k that does not satisfy d. Furthermore, r' is obtained from r by removing and/or duplicating tuples.*

Lemma 3.2 *If a relation r satisfies a set of constraint-generating k-dependencies $D = \{d_1, \ldots, d_m\}$ and does not satisfy a constraint-generating k_0-dependency d_0, then there is a relation r' of size k_0 that satisfies D but does not satisfy d_0.*

Lemma 3.3 *Consider an instance (D, d_0) of the dependency implication problem where d_0 is a constraint-generating k_0-dependency. The dependency d_0 is implied by D over all relations if and only if it is implied by D over relations of size k_0; i.e., $(\forall r)\Big[r \models D \Rightarrow r \models d_0\Big]$ iff $(\forall r')\Big[|r'| = k_0 \Rightarrow \Big[r' \models D \Rightarrow r' \models d_0\Big]\Big]$.*

The above lemmas generalize properties of uninterpreted dependencies.

3.2.2 Second-order Quantifier Elimination

By Lemma 3.3, in order to decide the implication problem, we just need to be able to decide this problem over relations of size k for a given k. Deciding the implication (1) thus reduces to deciding

$$(\forall r')\Big[[|r'| = k \wedge D(r')] \Rightarrow d_0(r')\Big]. \tag{2}$$

Let $r' = \{t_{x_1}, \ldots, t_{x_k}\}$ denote an arbitrary relation of size k where $t_{x_1}, \ldots, t_{x_k}$ are arbitrary tuples. We can eliminate the (second-order) quantification over relations from the implication (2) and replace it with a quantification over tuples (that is, over vectors of elements of the domain). We get

$$(\forall t_{x_1}) \cdots (\forall t_{x_k})\Big[D(\{t_{x_1}, \ldots, t_{x_k}\}) \Rightarrow d_0(\{t_{x_1}, \ldots, t_{x_k}\})\Big]. \tag{3}$$

3.2.3 Symmetrization

Next, we simplify the formula (3), whose validity is equivalent to the constraint dependency implication problem, by eliminating the quantification over tuples that appears within the dependencies of $D \cup \{d_0\}$. We refer to this quantifier elimination procedure for dependencies as *symmetrization*. For the sake of clarity, we present the details of the symmetrization process for the case where all dependencies are 1-dependencies or 2-dependencies and where d_0 is a 2 dependency, which implies that the implication problem can be solved over relations of size 2, i.e. $k = 2$. The process can be extended directly to the more general case.

For the case where $k = 2$, the formula (3) to be decided is the following.

$$(\forall t_x)(\forall t_y)\Big[D(\{t_x, t_y\}) \Rightarrow d_0(\{t_x, t_y\})\Big].$$

We can simplify this formula further by eliminating the quantification over tuples that appears in the dependencies $d(\{t_x, t_y\})$ in $D \cup \{d_0\}$. Every such dependency $d(\{t_x, t_y\})$ can indeed be rewritten as a constraint formula $cf_2(d)$ in the following manner (the subscript 2 in cf_2 recalls the fact that we are applying symmetrization in the context of a reduction to implication over 2-tuple relations).

1. Let d be a 1-dependency, that is, d is of the form $(\forall t)\Big[[r'(t) \wedge C[t]] \Rightarrow C'[t]\Big]$. This dependency considered over $r' = \{t_x, t_y\}$ is equivalent to the constraint formula

   $$cf_2(d): \quad \Big[C[t_x] \Rightarrow C'[t_x]\Big] \wedge \Big[C[t_y] \Rightarrow C'[t_y]\Big],$$

   which is a conjunction of $k = 2$ constraint implications. Notice that the t_x and t_y appearing in this formula are just tuples of variables ranging over the domain of the constraint theory of interest.

2. Let d be a 2-dependency, that is, d is of the form

   $$(\forall t_1)(\forall t_2)\Big[[r'(t_1) \wedge r'(t_2) \wedge C[t_1, t_2]] \Rightarrow C'[t_1, t_2]\Big].$$

   This dependency considered over $r' = \{t_x, t_y\}$ is equivalent to the constraint formula

   $$\begin{aligned} cf_2(d): \quad & \Big[C[t_x, t_y] \Rightarrow C'[t_x, t_y]\Big] \wedge \Big[C[t_y, t_x] \Rightarrow C'[t_y, t_x]\Big] \wedge \\ & \Big[C[t_x, t_x] \Rightarrow C'[t_x, t_x]\Big] \wedge \Big[C[t_y, t_y] \Rightarrow C'[t_y, t_y]\Big], \end{aligned}$$

   which is a conjunction of $k^k = 4$ constraint implications.

The rewriting of d as $cf_2(d)$ is what we call the *symmetrization* of d, for rather obvious reasons. It extends directly to any value of k. Notice that for a given k, any j-dependency d is rewritten as a constraint formula $cf_k(d)$, which is a conjunction of

k^j constraint implications. Interestingly, in the case of functional dependencies, symmetrization degenerates and produces only a single constraint implication. This is due to the fact that the underlying constraints are equalities, which are already symmetric, and to the special form of functional dependencies. Hence, in that special case, besides trivial formulas, symmetrization would only produce multiple instances of the same constraint formula.

Applying the symmetrization process to all the dependencies appearing in the formula (3), we get

$$(\forall t_{x_1}) \cdots (\forall t_{x_k}) \Big[cf_k(d_1) \wedge \cdots \wedge cf_k(d_m) \Rightarrow cf_k(d_0) \Big]. \tag{4}$$

Notice that in formula (4), each tuple variable can be replaced by n domain variables, and thus the quantification over tuples can be replaced by a quantification over elements of the domain. For the sake of clarity, we simply denote by $(\forall *)$ the adequate quantification over elements of the domain (the *universal closure*). Formula (4) thus becomes

$$(\forall *) \Big[cf_k(d_1) \wedge \cdots \wedge cf_k(d_m) \Rightarrow cf_k(d_0) \Big], \tag{5}$$

where each $cf_k(d)$ is a conjunction of k^j constraint implications if d is a j-dependency and d_0 is a k-dependency. Thus, we have proved the following theorem.

Theorem 3.4 *For constraint-generating k-dependencies, with bounded k, the implication problem is linearly reduced to the validity of a universally quantified formula of the constraint theory.*

Example 3.1 Let us consider the following constraint-generating 2-dependencies over a relation r with a single attribute.

$$\begin{array}{ll} d_1: & (\forall x)(\forall y) \Big[r(x) \wedge r(y) \Rightarrow x \leq y \Big] \\ d_2: & (\forall x)(\forall y) \Big[r(x) \wedge r(y) \Rightarrow x = y \Big] \end{array}$$

Symmetrizing them produces the following constraint formulas.

$$\begin{array}{ll} cf_2(d_1): & x \leq y \wedge y \leq x \wedge x \leq x \wedge y \leq y \\ cf_2(d_2): & x = y \wedge y = x \wedge x = x \wedge y = y \end{array}$$

It is clear that these two constraint formulas are equivalent, as they should be. ∎

We should point out that the implication problem for constraint-generating dependencies requires moving beyond purely Horn reasoning, as should be clear from the following example.

Example 3.2 Consider the following dependencies over a relation r with two attributes A and B.

$$\begin{aligned}
d_3 &: (\forall t_1)(\forall t_2)\Big[[r(t_1) \wedge r(t_2) \wedge t_1[A] \leq t_2[A]] \Rightarrow t_1[B] = t_2[B]\Big] \\
d_4 &: (\forall t_1)(\forall t_2)\Big[[r(t_1) \wedge r(t_2) \wedge t_1[A] \geq t_2[A]] \Rightarrow t_1[B] = t_2[B]\Big] \\
d_5 &: (\forall t_1)(\forall t_2)\Big[[r(t_1) \wedge r(t_2)] \Rightarrow t_1[B] = t_2[B]\Big]
\end{aligned}$$

The set $\{d_3, d_4\}$ implies d_5 because the set of formulas (implications)

$$\Big\{t_1[A] \leq t_2[A] \Rightarrow t_1[B] = t_2[B], \quad t_1[A] \geq t_2[A] \Rightarrow t_1[B] = t_2[B]\Big\}$$

implies $t_1[B] = t_2[B]$. But this conclusion requires a type of reasoning that can handle case analysis, which is beyond the scope of Horn reasoning. ∎

4 Complexity Results

4.1 Clausal dependencies

In this section, we study the complexity of the implication problem for some classes of constraint-generating dependencies occurring in practice, in particular dependencies with equality, order, and arithmetic constraints. We restrict our attention to atomic constraints and clausal dependencies as defined below.

Definition 4.1 An *atomic constraint* is a formula consisting of an interpreted predicate symbol applied to terms. A *clausal formula* is a conjunction of disjunctions of atomic constraints. A *clausal* constraint-generating dependency is a constraint-generating dependency such that the constraint in the antecedent is a conjunction of atomic constraints and the constraint in the consequent is an atomic constraint.

Notice that a constraint-generating dependency in which the constraint in the antecedent and the constraint in the consequent are both conjunctions of atomic constraints can be rewritten as a set of clausal constraint-generating dependencies (by decomposing the conjunction in the consequent). Essentially all the dependencies mentioned in [23] can be written in clausal form.

Moreover, we assume that the constraint language is *closed under negation*, i.e., the negation of an atomic constraint of the language is also a basic predicate of the constraint language.[3] This is again satisfied by many examples of interest, the most notable exception being the class of functional dependencies. We start our study with classes of k-dependencies for fixed values of k (mainly $k = 2$). This makes it possible to contrast our results with the results about functional dependencies which are 2-dependencies and for which the implication problem can be solved in $O(n)$. We then examine how letting k vary impacts our results.

We proceed by reducing clausal dependency implication to unsatisfiability of clausal formulas. More precisely, we negate the result of the symmetrization (i.e., formula 5) to obtain

$$(\exists *)\Big[cf_k(d_1) \wedge \cdots \wedge cf_k(d_m) \wedge \neg cf_k(d_0)\Big]. \tag{6}$$

For a clausal k-dependency d_0, $cf_k(d_0)$ is a conjunction of k^k clauses of the size of d_0. Thus $\neg cf_k(d_0)$ is a disjunction of k^k conjunctions. We thus split formula (6) into k^k formulas of the form

$$\Psi = (\exists *)\Big[\bigwedge_i \Big(\bigvee_j (c_{ij})\Big)\Big], \tag{7}$$

where each c_{ij} is an atomic constraint and where, if $|D| = m$, the number of clauses is at most equal to $m \cdot k^k$ plus the number of constraints in d_0. The number of literals in each clause is equal to the number of atomic constraints in the dependencies of D, or to 1 for the clauses obtained from the decomposition of d_0. Thus deciding the validity of the implication problem for k-dependencies (k fixed) can be done by checking the unsatisfiability of k^k conjunctions of clauses of length that is linear in the size of $D \cup \{d_0\}$. We can replace the variables in the constraint formulas by the corresponding Skolem constants and view the formulas Ψ as ground formulas.

The opposite LOGSPACE reduction, from unsatisfiability of clausal formulas to implication, also exists and requires only 1-dependencies. Assume we are given a clausal formula of the constraint language

$$\Psi = \bigwedge_{1 \le i \le p} \Big(\bigvee_{1 \le j \le q_i} (c_{ij}) \Big)$$

over n variables $x_1, \ldots, x_n$. We construct a set of clausal dependencies D_Ψ in the following way: for every conjunct $\bigvee_{1 \le j \le q_i}(c_{ij})$, $1 \le i \le p$ of Ψ, D_Ψ contains a dependency d_i of the

[3]Note that in this context, the distinction between positive and negative atomic constraints is meaningless.

form

$$r(x_1, \ldots, x_n), \neg c_{i,1}, \ldots, \neg c_{i,q_i-1} \Rightarrow c_{i,q_i}.$$

Note that since the constraint theory is closed under negation, the negations of atomic constraints are also atomic constraints. Finally, the dependency d_0 is chosen to be

$$r(x_1, \ldots, x_n) \Rightarrow A$$

where A is any unsatisfiable constraint in the domain theory. Clearly, Ψ is unsatisfiable iff D_Ψ implies d.

4.2 Equality and order constraints

We consider here atomic constraints of the form $x\theta y$ where $\theta \in \{=, \neq, <, \leq\}$ over integers, rationals, or reals.[4] This constraint language has two sublanguages closed under negation which we also study: $\{=, \neq\}$-constraints and $\{<, \leq\}$-constraints. We make the additional assumption that *no domain constants appear in the dependencies.* (If this assumption is not satisfied, the complexity usually shifts up. For example, in Theorem 4.1 the first case becomes co-NP-complete for the integers by the results of [34].) Finally, our results assume that we are dealing with k-dependencies for a fixed k, but that the database schema, i.e. the number of attributes and hence the number of available variables in k-dependencies, can vary.

Theorem 4.1 *The implication problem for clausal constraint-generating k-dependencies is:*

1. *in PTIME for dependencies with one atomic $\{=, \neq, <, \leq\}$-constraint (no constraints in the antecedent),*
2. *co-NP-complete for dependencies with two or more atomic $\{=, \neq\}$-constraints,*
3. *co-NP-complete for dependencies with two or more atomic $\{<, \leq\}$-constraints.*

Proof: The first result follows from [42, page 892]. (For recent efficient algorithms for this problem, see [38, 20].) The membership in co-NP for the two remaining cases

[4] In fact, our lower bounds hold for any infinite linearly-ordered set.

follows from the fact that checking the satisfiability of a conjunction of equality and order constraints can be done in polynomial time.

To prove the lower bounds, we reduce an NP-complete problem to satisfiability of a set of ground clauses with at most two literals corresponding to the formula Ψ in equation (7) above. This reduction is then composed with the reduction from unsatisfiability to dependency implication.

For $\{=, \neq\}$-constraints, we use a reduction from GRAPH-3-COLORABILITY. For a graph with n vertices, we need $2n+2$ Skolem constants: a, b, $a_1, \ldots, a_n$, $b_1, \ldots, b_n$. The idea is to use the pair (a_i, b_i) to encode the color of the vertex i: (a, a) stands for 1, (a, b) for 2, and (b, a) for 3. For every vertex i, we have the following clauses: $(a_i = a \vee a_i = b)$, $(b_i = a \vee b_i = b)$, and $(a_i = a \vee b_i = a)$. For every edge (i, j), we have the clause $(a_i \neq a_j \vee b_i \neq b_j)$. Finally, there is a clause $a \neq b$.

For $\{<, \leq\}$-constraints, we use a reduction from BETWEENNESS [14, page 279]: given a finite set A (of n elements) and a collection S of ordered triples (a, b, c) of distinct elements from A, determine whether there is a 1-1 function $f : A \to \{1, \ldots, n\}$ such that for each $(a, b, c) \in S$, we have either $f(a) < f(b) < f(c)$ or $f(c) < f(b) < f(a)$.

The set A is represented as the set of indices $\{1, \ldots, n\}$ and the collection S accordingly. The Skolem constants are: $a_1, \ldots, a_n$. For every $i \neq j$, we have the clause $(a_i < a_j \vee a_j < a_i)$ to encode that f is 1-1. For every $(i, j, \ell) \in S$, we have $(a_i < a_j \wedge a_j < a_\ell) \vee (a_\ell < a_j \wedge a_j < a_i)$. The last formula can be rewritten as four 2-literal clauses. This reduction encodes a 1-1 function from A onto $\{x_1, \ldots, x_n\}$, an n-element subset of the domain. Because the domain is linearly ordered, a 1-1 function f from A to $\{1, \ldots, n\}$ can be defined as

$$f(i) = \text{index of } x_i \text{ in } \{x_1, \ldots, x_n\}.$$

Note that it is enough for f to be uniquely defined. It may be the case that the construction of f itself is very hard, possibly even non-recursive, for some linearly-ordered domains. ∎

Notice that we only use two literals per clause, whereas a propositional encoding of these problems would require three literals per clause. Notice also that we have been assuming infinite domains. For finite domains of size greater than 2, the implication problem is co-NP-complete even for dependencies with one atomic constraint. For domains of size 2, the implication is in PTIME by an easy reduction to 2-SAT.

The above results are rather negative. To obtain more tractable classes, we propose to further restrict the syntax of dependencies by typing.

Definition 4.2 A clausal dependency is *typed* if each atomic constraint involves only the values of one given attribute in different tuples.

The second dependency in Example 2.1 of Section 2 (i.e., the property of r being "globally nondecreasing") is typed, while the first one (the property of r being "strongly retroactively bounded") and the dependency of Example 2.2 are not. Functional dependencies are also typed.

Notice that for typed dependencies, the reduction from unsatisfiability to dependency implication given above is not useful for obtaining lower bounds. Indeed, it reduces unsatisfiability to implication of 1-dependencies which are not typed. Furthermore, this reduction cannot in general be adapted to yield typed 2-dependencies. Indeed, because of the symmetrization procedure, the constraint problem obtained from typed 2-dependencies has a particular symmetric structure (for 1-dependencies, there is no symmetrization). The question thus is whether this symmetric structure is sufficient for lowering the complexity of the constraint problem that has to be solved. As shown in the following theorem, the answer is fortunately positive.

Theorem 4.2 *The implication problem for typed clausal constraint-generating* 2*-dependencies with at most two atomic* $\{=, \neq, <, \leq\}$*-constraints is in PTIME* $(O(n))$*.*

Proof: A typed 2-dependency is of the form

$$(\forall t_x)(\forall t_y)\Big[[r(t_x) \wedge r(t_y) \wedge (t_x[i] \; pred_\ell \; t_y[i])] \Rightarrow (t_x[j] \; pred_r \; t_y[j])\Big] \tag{8}$$

where each of $pred_\ell$ and $pred_r$ is one of $\{=, \neq, <, \leq\}$. By Lemma 3.3, the implication problem for typed 2-dependencies coincides with the implication problem over 2-tuple relations. The remaining steps of the reduction given in Section 3 show how this implication can be reduced to a pure constraint problem. However, since we need to take into account the specific nature of the constraint problem obtained for typed 2-dependencies, our starting point for the proof of this theorem is further upstream. We consider the problem of deciding whether for a typed 2-dependency d_0 and a set D of dependencies of the same kind, $D \models d_0$ over 2-tuple relations, or equivalently whether $D \wedge \neg d_0$ is unsatisfiable over 2-tuple relations. We give a PTIME algorithm for deciding satisfiability (and hence unsatisfiability) over 2-tuple relations of $D \wedge \neg d_0$.

Among the predicates in $\{=, \neq, <, \leq\}$, we distinguish the set *eq-pred* : $\{=, \leq\}$, and the set *diff-pred* : $\{\neq, <\}$. The intuition is that members of *eq-pred* can be satisfied when their arguments are equal, whereas members of *diff-pred* cannot be satisfied in that case. This allows us to define four classes of constraint dependencies:

$$\begin{array}{lllr} \textit{eq-pred} & \Rightarrow & \textit{eq-pred} & (9) \\ \textit{eq-pred} & \Rightarrow & \textit{diff-pred} & (10) \end{array} \qquad \begin{array}{lllr} \textit{diff-pred} & \Rightarrow & \textit{eq-pred} & (11) \\ \textit{diff-pred} & \Rightarrow & \textit{diff-pred} & (12) \end{array}$$

Notice that (10) and (11) are self-contrapositives, whereas (9) and (12) are each other's contrapositives. We thus only need one of the latter two categories and choose to keep (12). Furthermore, all dependencies of the form (10) are unsatisfiable (over nonempty relations). Indeed, if in (8) one chooses $t_x = t_y$, then ($t_x[i]$ *eq-pred* $t_x[i]$) is true whereas ($t_x[j]$ *diff-pred* $t_y[j]$) has to be false and the implication is false. Thus, if such a dependency occurs in D, this set is trivially unsatisfiable and we can assume without loss of generality that D only contains dependencies of the forms (11) and (12). Similarly, if d_0 is of the form (10), $\neg d_0$ is valid and, since D is always satisfiable by a one tuple relation if it does not contain dependencies of the form (10), $D \wedge \neg d_0$ is satisfiable. We can thus also assume without loss of generality that d_0 is either of the form (11) or of the form (12).

Since the dependencies are typed, each dependency d involves two attributes of the relation r which we refer to as l_d (the one on the left of the implication) and r_d (the one on the right of the implication). We are looking for a 2-tuple model of $D \wedge \neg d_0$. The first step of the procedure is to partition the attributes of the relation r into the set of those that must have a different value in the two tuples of the relation and those that may have the same value. We call the first *diff*-attributes and the second *eq*-attributes. The set *DA* of *diff*-attributes is obtained by the following procedure.

The initial extension of *DA* is obtained from d_0. If d_0 is of the form *diff-pred* $\Rightarrow$ *eq-pred*, then *DA* initially contains both l_{d_0} and r_{d_0}; whereas if d_0 is of the form *diff-pred* $\Rightarrow$ *diff-pred*, then initially $DA = \{l_{d_0}\}$. One then repeatedly applies the following step until saturation: if there is a dependency d of the form *diff-pred* $\Rightarrow$ *diff-pred* such that the attribute l_d is in *DA*, then the attribute r_d is added to *DA*. This procedure is similar to the one computing the closure of a set of attributes under a set of functional dependencies and hence can be implemented in linear time. From now on, let *DA* be the set of attributes obtained by this procedure.

A direct consequence of the way in which DA is constructed is that any 2-tuple model in which both tuples give the same value to some attributes in DA cannot satisfy $D \wedge \neg d_0$. Furthermore, we claim that if $D \wedge \neg d_0$ has a 2-tuple model, it has a 2-tuple model in which all attributes in DA have different values in the two tuples; and all attributes not in DA have the same value in both tuples. To prove this, assume there is a model and give an arbitrary identical value in both tuples to the attributes not in DA. Since dependencies of the form 12 cannot have their attribute r_d out of DA without also having their attribute l_d out of DA, this can only change the truth value of the dependencies in D and of $\neg d_0$ from *false* to *true*, and hence we still have a model.

Thus, in order to find a model for $D \wedge \neg d_0$, it is sufficient to find values for the attributes in DA. We know that these values have to be different and, given the restrictions on the theory we are working within, the only relevant property of these values is their order (which one is smaller than the other). Let us call the two possible orders u (up) and d (down). The choice between u and d for each attribute i can be encoded by one boolean proposition $u[i]$ (*true* if the order for i is u, *false* if it is d). The problem thus is to find truth values for the propositions $u[i]$ in such a way that they define a model of $D \wedge \neg d_0$.

To do this, we encode the conditions imposed by the dependencies referring to attributes that are both in DA. Indeed, for dependencies in D (and for $\neg d_0$), if one of the atomic constraints does not refer to an attribute in DA, the dependency ($\neg d_0$) is satisfied whatever the order chosen for the attributes.

We construct the constraints on the propositions u for dependencies in positive form as they appear in D. For $\neg d_0$, one applies the construction to d_0 and negates the result. There are 9 cases of dependencies of the form *diff-pred* $\Rightarrow$ *diff-pred*:

$$\begin{array}{llll@{\qquad}llll@{\qquad}llll}
\neq & \Rightarrow & \neq & (13) & < & \Rightarrow & \neq & (16) & > & \Rightarrow & \neq & (19) \\
\neq & \Rightarrow & < & (14) & < & \Rightarrow & < & (17) & > & \Rightarrow & < & (20) \\
\neq & \Rightarrow & > & (15) & < & \Rightarrow & > & (18) & > & \Rightarrow & > & (21)
\end{array}$$

Cases 13, 16, and 19 translate to *true* (we have imposed that attributes in DA have different values in both tuples). Cases 14 and 15 are always unsatisfiable (by symmetry) and thus translate to the constraint *false*. Cases 17 and 21 translate to

$$(u[l_d] \Rightarrow u[r_d]) \wedge (\neg u[l_d] \Rightarrow \neg u[r_d])\,,$$

whereas cases 18 and 20 translate to

$$(u[l_d] \Rightarrow \neg u[r_d]) \wedge (\neg u[l_d] \Rightarrow u[r_d])\,.$$

There are also 9 cases of dependencies of the form *diff-pred* $\Rightarrow$ *eq-pred*:

$$\begin{array}{lllllllllllll}
\neq & \Rightarrow & = & (22) & \quad & < & \Rightarrow & = & (25) & \quad & > & \Rightarrow & = & (28) \\
\neq & \Rightarrow & \leq & (23) & & < & \Rightarrow & \leq & (26) & & > & \Rightarrow & \leq & (29) \\
\neq & \Rightarrow & \geq & (24) & & < & \Rightarrow & \geq & (27) & & > & \Rightarrow & \geq & (30)
\end{array}$$

Cases 22, 23, 24, 25, and 28 are contradictory and translate to *false*. Cases 26 and 30 translate as 17 and 21 and, similarly, 27 and 29 are translated as 18 and 20.

The result of this encoding is a set of Boolean clauses with at most two literals per clause. Deciding if it is satisfiable can thus be done with the 2-SAT procedure which is in PTIME ($O(n)$) [1]. ∎

Theorem 4.3 *The implication problem for typed clausal constraint-generating* 2*-dependencies is:*

1. *co-NP-complete for dependencies with three or more atomic* $\{=, \neq\}$*-constraints,*
2. *co-NP-complete for dependencies with three or more atomic* $\{<, \leq\}$*-constraints.*

Proof: Proving the lower bounds in the typed case is more difficult than in Theorem 4.1 because the reverse reduction, from unsatisfiability of ground clauses to dependency implication that uses 1-dependencies, is not available. We can continue, however, to work with ground clauses as in the proof of Theorem 4.1 provided the clauses can be mapped back to typed 2-dependencies.

The proofs in both cases involve a reduction from SET SPLITTING [14, page 221]: given a collection S of 3-element subsets of a finite set U, determine whether there is a disjoint partition of U into two sets A and $U - A$ such that no set in S consists of elements only from A or only from $U - A$.

The proof thus proceeds in two steps. We first reduce SET SPLITTING to a collection of ground clauses C; then we show how to construct an instance of the implication problem $D \models d_0$ for typed 2-dependencies such that the set of clauses Ψ (see formula 7) obtained for this instance is equisatisfiable with C.

Let us consider first $\{=, \neq\}$-constraints. We let $U = \{x_1, \ldots, x_n\}$. We use $2n$ Skolem constants $a_1, \ldots, a_n$, $b_1, \ldots, b_n$, and for $i = 1, \ldots, n$, we represent the fact that x_i is in A by $a_i = b_i$ and the opposite situation by $a_i \neq b_i$. Now, for every set in S consisting of x_i, x_j, and x_k (at most three elements), we have the following two clauses.

$$a_i = b_i \vee a_j = b_j \vee a_k = b_k \tag{31}$$

$$a_i \neq b_i \vee a_j \neq b_j \vee a_k \neq b_k \tag{32}$$

The resulting set of clauses is satisfiable iff U has the desired partition. We have to check now whether the above clauses can be obtained as a result of the symmetrization procedure.

Let us assume that we deal with a relation r with n attributes and that we use Skolem constants $a_1, \ldots, a_n$ for the variables referring to elements of the first tuple, and Skolem constants $b_1, \ldots, b_n$ for the variables referring to elements of the second tuple used in the symmetrization procedure. A clause such as (31) is what is obtained from a typed dependency in D of the form:

$$(\forall t_x)(\forall t_y)\Big[[r(t_x) \wedge r(t_y) \wedge (t_x[i] \neq t_y[i]) \wedge (t_x[j] \neq t_y[j])] \Rightarrow (t_x[k] = t_y[k])\Big].$$

However, a clause such as (32) can only be obtained by considering D and d_0 together. It cannot be obtained by symmetrizing a single dependency $d \in D$ alone, because we would also obtain an unsatisfiable clause by instantiating t_x and t_y to the same tuple.

The idea is to cook up clauses of the form (32) from other "good" clauses. We need the following ingredient clauses. (There is one new pair of Skolem constants a_m and b_m for each clause of the form (32), and a single pair of new Skolem constants a_p and b_p that can be shared among all ingredient clauses.)

$$a_i \neq b_i \vee a_m \neq b_m \vee a_p = b_p \tag{33}$$

$$a_p \neq b_p \tag{34}$$

$$a_m = b_m \vee a_j \neq b_j \vee a_k \neq b_k \tag{35}$$

The restriction of every valuation that makes the set of ingredient clauses true makes also (32) true. And vice versa: every valuation that makes (32) true can be extended to a valuation that makes the set of ingredient clauses true.

It remains to be shown how to get the ingredient clauses. If there are k clauses of the form (32), we consider a relation with $n+k+1$ attributes. The ingredient clauses of the form (33) and (35) are not problematic because they can be obtained by including the appropriate functional dependencies in D. The ingredient clause (34) is obtained from the negation of $cf(d_0)$, where d_0 is the dependency $(\forall t_x)\forall t_y)\Big[[r(t_x) \wedge r(t_y)] \Rightarrow t_x[p] = t_y[p]\Big]$.

Let us consider now $\{<, \leq\}$-constraints. We represent the fact that x_i is in A by $a_i < b_i$ and the opposite situation by $b_i < a_i$. Now, for every set in S consisting of x_i, x_j, and x_k (at most three elements), we have the following clauses.

$$a_i \leq b_i \vee a_j \leq b_j \vee a_k \leq b_k \tag{36}$$

$$b_i \leq a_i \vee b_j \leq a_j \vee b_k \leq a_k \tag{37}$$

Additionally we force a_i and b_i, for all i, to be not equal by the clause

$$a_i \neq b_i. \tag{38}$$

The resulting set of clauses is satisfiable iff U has the desired partition. The clauses (36) and (37) are obtained directly from constraint dependencies. Inequality constraints (38) are manufactured as follows. First, notice that every unary functional dependency can be represented as a set of typed 2-dependencies with three $\{<, \leq\}$-constraints. From such a functional dependency, we can obtain a set of clauses that is equivalent to $(a_i \neq b_i \vee a_m = b_m)$, for some new Skolem constants a_m and b_m. These clauses together with $a_m \neq b_m$ yield the effect of having the clause $a_i \neq b_i$. The clause $a_m \neq b_m$ can be obtained from $\neg cf(d_0)$ as in the previous proof (using $\leq$ instead of $=$ yields the same dependency because of the symmetry).

It should be clear that the size of the set of constraints C and the corresponding sets of constraint dependencies are polynomial in the size of the instance of SET SPLITTING and can be obtained in LOGSPACE for both constraint languages considered. ∎

Theorem 4.2 yields a new class of dependencies with a tractable implication problem. This class properly contains that of unary functional dependencies and is incomparable with the class of all functional dependencies. Together, Theorems 4.1, 4.2 and 4.3 give a *complete classification* of tractable and intractable classes of untyped and typed 2-dependencies with $\{=, \neq, <, \leq\}$-constraints. The case of typed k-dependencies ($k > 2$) with two $\{=, \neq, <, \leq\}$-constraints is open. (The implication problem for such dependencies with three constraints is clearly co-NP-complete by Theorems 4.1 and 4.3.)

4.3 Linear arithmetic constraints

We consider now *linear arithmetic constraints*, i.e., atomic constraints of the form $a_1x_1 + \cdots + a_kx_k \leq a$ (domain constants are allowed here). We can use here directly the results about the complexity of linear programming [35].

Theorem 4.4 *For linear arithmetic constraints, the implication problem for clausal constraint-generating k-dependencies with one atomic constraint per dependency is in PTIME for the reals, and co-NP-complete for the integers.*

Proof: It is easy to see that the formula 7 represents in this case a linear programming problem. ∎

4.4 The impact of k on the complexity

It is quite natural to ask what our complexity results would become if one allowed k to vary. The question is mostly of theoretical interest (it is hard to think of naturally occurring dependencies that are not 2- or 3-dependencies), but leads to interesting observations.

Let us first see what impact letting k vary has on our PTIME results for clausal dependencies. In Theorem 4.1 case 1, the dependencies must be at most 2-dependencies since they each involve only one binary predicate. The same result thus trivially holds when k is allowed to vary. Theorem 4.2 is not just restricted to a fixed k, but to 2-dependencies. Letting k be part of the input rather than a fixed parameter thus makes no sense in this case. Finally, in the case of linear arithmetic constraints over the reals, letting k vary leads to a linear programming problem of size that is exponential in k and the PTIME result thus fails in this case, even for dependencies with a single atomic constraint.

In the construction we have given, k clearly has an exponential impact on the size of the constraint problem to be solved. So, it is natural to expect that in the cases where k is allowed to vary, the complexity would shoot up by one exponential, e.g. from NP to NEXPTIME. Fortunately, the situation is not that bad. After the second order quantifier elimination, we have to solve a $\forall * \exists *$ constraint validity problem. Indeed, the elimination of the quantification on relations introduces a universal quantification on domain elements and the quantification on tuples within the dependencies becomes existential since it is negated by the implication. Our costly symmetrization step thus aims at reducing a $\forall * \exists *$ constraint validity problem to a $\forall *$ validity problem, which in the cases we consider is in co-NP. Furthermore, in the case of order constraints, this quantifier elimination can be achieved much more efficiently as described in [27]. This implies that letting k vary only moves up the complexity one level up in the polynomial hierarchy, i.e. from co-NP to Π_2^P.

5 Conclusions and Related Work

A brief summary of this paper is that constraint-generating dependencies are an interesting concept, and that deciding implication of such dependencies is basically no harder than deciding the underlying constraint theory, which, a priori, was not obvious. The obvious applications of constraint-generating dependencies are constraint database design theory and consistency checking. Apart from the constraint languages considered in this paper, other languages may be relevant as well, for instance the *congruence constraints* that appear in [23, 24]. Also, the impact that the presence of domain constants in equality and order constraints has on the complexity of implication should be fully studied.

As far as related work, we should first mention that Jensen and Snodgrass [23] induced us to think about constraint dependencies. We should note that the integrity constraints over temporal databases postulated there involve both typed and untyped constraint-generating dependencies, as well as tuple-generating ones.

Two other relevant papers on *implication constraints* by Ishakbeyoğlu, Ozsoyoğlu and Zhang [22, 44], as well as a paper on efficient integrity checking by Gupta, Sagiv, Ullman, and Widom [21] contain work fairly close to ours. However, there are several important differences. Foremost, all three papers discuss a fixed language of constraint formulas, namely equality ($=$), inequality ($\neq$), and order ($<, \leq$) constraints, while our results are applicable to any decidable constraint theory thanks to our general reduction strategy. In particular, the papers [44, 21], which were written independently of the first version of this paper, both present results equivalent to our Theorem 3.4, but formulated in the context of a fixed constraint language. Also, the proof techniques in those papers, based on the theory of conjunctive queries, are quite different from ours. Moreover, the complexity results of [44] are obtained in a slightly different model. Both the number of database literals and the arity of relations in a dependency are considered as parts of the input, while we consider only the latter. We think that our model is more intuitive because it is difficult to come up with a meaningful dependency that references more than a few tuples in a relation. Our intractability results are stronger than those of [44] while our positive characterizations of polynomial-time decidable problems do not necessarily carry over to the framework of [44]. Also, in [22, 44], the tractable classes of dependencies are not defined syntactically but rather by the presence or absence of certain types of refutations.

A clausal constraint-generating dependency (quantifiers omitted)

$$r(t_1) \wedge \cdots \wedge r(t_k) \wedge C_1 \wedge \cdots \wedge C_n \Rightarrow C_0$$

can be viewed as an integrity constraint (in the notation of [21])

$$\textbf{panic} \; :- \; r(t_1) \,\&\, \cdots \,\&\, r(t_k) \,\&\, C_1 \,\&\, \cdots \,\&\, C_n \,\&\, \neg C_0.$$

Thus the implication of a dependency by a set of dependencies is equivalent to the subsumption of an integrity constraint by a set of integrity constraints. Therefore the results about the complexity of implication from Section 4 transfer directly to the context of constraint subsumption. The paper [21] applies the results about constraint subsumption to develop techniques for efficient integrity checking. Unfortunately, this application requires introducing constants into constraints, so our complexity results, developed under the assumption that constants do not appear in dependencies, are not applicable here, though our general reduction is.

Order dependencies, proposed by Ginsburg and Hull [15, 16], are typed clausal 2-dependencies over the theory of equality and order (without $\neq$). The order is not required to be total. Ginsburg and Hull provided an axiomatization of such dependencies and proved that the implication problem is co-NP-complete for dependencies with at least three constraints. To prove the lower bound they used, however, dependencies with equality and order constraints, while we proved the lower bounds for both theories separately (Theorem 4.3). Ginsburg and Hull also supplied a number of tractable dependency classes which are, again, different from ours and involve mainly partial orders.

Maher [30] considered constrained extensions of functional dependencies and of finiteness dependencies, which may be of interest in the analysis and optimization of CLP programs. These are functional (or finiteness) dependencies that hold only of those tuples in a relation that satisfy the given constraint. Maher addresses the implication problem for such constrained dependencies by providing axiomatic proof systems and algorithms for computing the closure of a set of dependencies. In the case of constrained functional dependencies, he proves that the implication problem is in PTIME when the constraint theory satisfies a property called *independence of negative constraints* and the constraint implication problem is solvable in PTIME. Maher's constrained functional dependencies are actually a special case of our constraint-generating dependencies. His PTIME complexity result is obtained under different restrictive hypotheses than ours. It should be noted that the independence of negative constraints condition is only really meaningful in the context of equality constraints, and does not hold for constraint languages closed under negation, such as order constraints or linear arithmetic constraints.

Other forms of constraint dependencies can also be of interest. An obvious candidate is the concept of *tuple-generating* constraint dependency. in order to solve the implication problem for such dependencies, the chase procedure needs to be appropriately generalized. Maher and Srivastava [31] have proposed two different generalizations of the chase algorithm for constraint tuple-generating dependencies that are shown to be equivalent when the constraint theory satisfies the independence of negative constraints.

References

[1] B. Aspvall, M. Plass, and R. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Inf. Process. Lett.*, 8(3):121–123, 1979.

[2] M. Baudinet. Temporal logic programming is complete and expressive. In *Sixteenth ACM Symposium on Principles of Programming Languages*, pages 267–280, Austin, Texas, Jan. 1989.

[3] M. Baudinet. On the expressiveness of temporal logic programming. *Information and Computation*, 117(2):157–180, Mar. 1995.

[4] M. Baudinet, J. Chomicki, and P. Wolper. Temporal deductive databases. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases. Theory, Design, and Implementation*, chapter 13, pages 294–320. Benjamin/Cummings, 1993.

[5] M. Baudinet, M. Niézette, and P. Wolper. On the representation of infinite temporal data and queries. In *Tenth ACM Symposium on Principles of Database Systems*, pages 280–290, Denver, Colorado, May 1991.

[6] C. Beeri and M. Vardi. A proof procedure for data dependencies. *Journal of the ACM*, 31(4):718–741, Oct. 1984.

[7] M. Benedikt, G. Dong, L. Libkin, and L. Wong. Relational Expressive Power of Constraint Query Languages. *Journal of the ACM*, 45:1–34, 1998.

[8] A. Brodsky, J. Jaffar, and M. J. Maher. Towards Practical Constraint Databases. *Constraints*, 2(3/4):279–304, 1997.

[9] A. Brodsky, C. Lassez, and J.-L. Lassez. Separability of polyhedra for optimal filtering of spatial and constraint data. In *Fourteenth ACM Symposium on Principles of Database Systems*, San Jose, California, May 1995.

[10] J. Chomicki. Polynomial time query processing in temporal deductive databases. In *Ninth ACM Symposium on Principles of Database Systems*, pages 379–391, Nashville, Tennessee, Apr. 1990.

[11] J. Chomicki and T. Imieliński. Temporal deductive databases and infinite objects. In *Seventh ACM Symposium on Principles of Database Systems*, pages 61–73, Austin, Texas, Mar. 1988.

[12] J. Chomicki and T. Imieliński. Finite Representation of Infinite Query Answers. *ACM Transactions on Database Systems*, 18(2):181–223, June 1993.

[13] J. Cox and K. McAloon. Decision procedures for constraint based extensions of Datalog. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*. MIT Press, 1993.

[14] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

[15] S. Ginsburg and R. Hull. Order dependency in the relational model. *Theoretical Computer Science*, 26:149–195, 1983.

[16] S. Ginsburg and R. Hull. Sort sets in the relational model. *Journal of the ACM*, 33(3):465–488, July 1986.

[17] S. Grumbach and J. Su. First-order definability over constraint databases. In *Principles and Practice of Constraint Programming, First International Conference, CP'95*, pages 121–136, Cassis, France, Sept. 1995. LNCS 976, Springer.

[18] S. Grumbach and J. Su. Finitely Representable Databases. *Journal of Computer and System Sciences*, 55(2):273–298, 1997.

[19] S. Grumbach and J. Su. Queries with Arithmetical Constraints. *Theoretical Computer Science*, 173(1):151–181, 1997.

[20] S. Guo, W. Sun, and M.A. Weiss. Solving Satisfiability and Implication Problems in Database Systems. *ACM Transactions on Database Systems*, 21(2):270–293, 1996.

[21] A. Gupta, Y. Sagiv, J. D. Ullman, and J. Widom. Constraint checking with partial information. In *Thirteenth ACM Symposium on Principles of Database Systems*, pages 45–55, Minneapolis, MN, May 1994.

[22] N. S. Ishakbeyoğlu and Z. M. Ozsoyoğlu. On the maintenance of implication integrity constraints. In *Fourth International Conference on Database and Expert Systems Applications*, pages 221–232, Prague, Sept. 1993. LNCS 720, Springer.

[23] C. Jensen and R. Snodgrass. Temporal specialization and generalization. *IEEE Transactions of Knowledge and Data Engineering*, 6(6):954–974, 1994.

[24] F. Kabanza, J.-M. Stévenne, and P. Wolper. Handling infinite temporal data. *Journal of Computer and System Sciences*, 51(1):3–17, Aug. 1995.

[25] P. Kanellakis, G. Kuper, and P. Revesz. Constraint query languages. *Journal of Computer and System Sciences*, 51(1):26–52, Aug. 1995.

[26] P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter. Indexing for data models with constraints and classes. In *Twelfth ACM Symposium on Principles of Database Systems*, pages 233–243, Washington, DC, May 1993.

[27] M. Koubarakis. Complexity results for first-order theories of temporal constraints. In *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning (KR'94)*, pages 379–390, May 1994.

[28] M. Koubarakis. Database Models for Infinite and Indefinite Temporal Information. *Information Systems*, 19(2):141–174, March 1994.

[29] M. Koubarakis. The Complexity of Query Evaluation in Indefinite Temporal Constraint Databases. *Theoretical Computer Science*, 171(1–2):25–60, 1997.

[30] M.J. Maher. Constrained Dependencies. *Theoretical Computer Science*, 173(1):113–149, 1997.

[31] M. J. Maher and D. Srivastava. Chasing constrained tuple-generating dependencies. In *Fifteenth ACM Symposium on Principles of Database Systems*, Montreal, Canada, June 1996. ACM Press.

[32] D. Maier. *The Theory of Relational Databases.* Computer Science Press, 1983.

[33] P. Z. Revesz. A Closed-Form Evaluation for Datalog Queries with Integer (Gap)-Order Constraints. *Theoretical Computer Science*, 116:117–149, 1993.

[34] D. Rosenkrantz and H. B. I. Hunt. Processing conjunctive predicates and queries. In *International Conference on Very Large Data Bases*, pages 64–72, 1980.

[35] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, 1986.

[36] D. Srivastava. Subsumption and indexing in constraint query languages with linear arithmetic constraints. *Annals of Mathematics and Artificial Intelligence*, 1993.

[37] A. P. Stolboushkin and M. A. Taitslin. Linear vs. order constraint queries over rational databases. In *Fifteenth ACM Symposium on Principles of Database Systems*, pages 17–27, Montreal, Canada, June 1996. ACM Press.

[38] W. Sun and M.A. Weiss. An Efficient Algorithm for Testing Implication Involving Arithmetic Inequalities. *IEEE Transactions on Knowledge and Data Engineering*, 6(6):997–1001, 1994.

[39] B. Thalheim. *Dependencies in Relational Databases.* Teubner-Texte zur Mathematik, Band 126. B.G. Teubner Verlagsgesellschaft, Stuttgart, 1991.

[40] D. Toman and J. Chomicki. Datalog with Integer Periodicity Constraints. *Journal of Logic Programming*, 35(3):263–290, 1998.

[41] D. Toman. Memoing evaluation for constraint extensions of datalog. *Constraints*, 2(3/4):337–359, 1998.

[42] J. D. Ullman. *Principles of Database and Knowledge-Base Systems. Volume II: The New Technologies.* Computer Science Press, 1989.

[43] R. van der Meyden. The Complexity of Querying Indefinite Data about Linearly Ordered Domains. *Journal of Computer and System Sciences*, 54(1):113–135, 1997.

[44] X. Zhang and Z. M. Ozsoyoğlu. On efficient reasoning with implication constraints. In *Third International Conference on Deductive and Object-Oriented Databases*, Phoenix, Arizona, Dec. 1993.