Function Symbols in Tuple-Generating Dependencies: Expressive Power and Computability

Georg Gottlob
University of Oxford
georg.gottlob@cs.ox.ac.uk

Reinhard Pichler
Vienna University of Technology
pichler@dbai.tuwien.ac.at

Emanuel Sallinger
Vienna University of Technology
sallinger@dbai.tuwien.ac.at

ABSTRACT

Tuple-generating dependencies – for short tgds – have been a staple of database research throughout most of its history. Yet one of the central aspects of tgds, namely the role of existential quantifiers, has not seen much investigation so far. When studying dependencies, existential quantifiers and – in their Skolemized form – function symbols are often viewed as two ways to express the same concept. But in fact, tgds are quite restrictive in the way that functional terms can occur.

In this paper, we investigate the role of function symbols in dependency formalisms that go beyond tgds. Among them is the powerful class of SO tgds and the intermediate class of nested tgds. In addition, we employ Henkin quantifiers – a well-known concept in the area of logic – and introduce Henkin tgds to gain a more fine-grained understanding of the role of function symbols in dependencies.

For members of these families of dependency classes, we investigate their expressive power, that is, when one dependency class is equivalently representable in another class of dependencies. In addition, we analyze the computability of query answering under many of the well-known syntactical decidability criteria for tgds as well as the complexity of model checking.

Categories and Subject Descriptors

H.2 [**Database Management**]: General

General Terms

Theory; Algorithms

Keywords

Dependencies; Function Symbols; Expressive Power

PODS'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

ACM 978-1-4503-2757-2/15/05 ...$15.00.
http://dx.doi.org/10.1145/2745754.2745756 .

1. Introduction

Tuple-generating dependencies – for short tgds – have been a staple of database research throughout most of its history. Tgds appear under many different names in many different areas. They are often called existential rules in the area of artificial intelligence [5, 6]. In the area of data exchange [13] and integration [23], tgds are the most common types of dependencies used for formulating so-called schema mappings, which are high-level specifications of the relationship between two databases.

Yet one of the central aspects of tgds, namely the role of existential quantifiers, has not seen much investigation so far. When studying dependencies, existential quantifiers and – in their Skolemized form – function symbols are often seen as two ways to express the same concept. But in fact, tgds are quite restrictive in the way that functional terms can occur. Consider the following tgd based on employees, their departments and the department managers:

$$\forall e, d\ \ \mathsf{Emp}(e, d) \rightarrow \exists dm\, \mathsf{Mgr}(e, dm)$$

To understand the exact form of existential quantification, let us look at its Skolemized form, where the implicit dependence of the existential quantifier is made explicit using function symbols. That is, the variable dm is replaced by a term based on the function f_{dm}.

$$\exists f_{\mathsf{dm}}\, \forall e, d\ \ \mathsf{Emp}(e, d) \rightarrow \mathsf{Mgr}(e, f_{\mathsf{dm}}(e, d))$$

Observe that any functional term contains the full set of universally quantified variables from the antecedent. More concretely, the function f_{dm} representing the department manager depends on both the department and the employee.

In contrast, what we would probably like to express is that the department manager only depends on the department. That is, the dependency

$$\exists f_{\mathsf{dm}}\, \forall e, d\ \ \mathsf{Emp}(e, d) \rightarrow \mathsf{Mgr}(e, f_{\mathsf{dm}}(d))$$

This dependency cannot be expressed by a logically equivalent set of tgds[1]. However, there are more powerful dependency languages than tgds, most importantly SO tgds [14][2]. The key feature of SO tgds is the use

[1]In Section 4 we will show the stronger result that not even a relaxation of logical equivalence allows the simulation of a similar dependency by a set of tgds.

[2]Note that as originally defined, SO tgds are required to be so-called source-to-target (s-t) dependencies. In the context of this paper, we do not restrict any dependency formalism to s-t unless explicitly mentioned.

of function symbols, and indeed, the above formula is an SO tgd. SO tgds and their subclass plain SO tgds [4] were shown to be particularly suited for expressing composition and inversion of schema mappings, two key operators for schema mapping management [7]. For recent surveys on the many problems studied for schema mappings based on tgds and SO tgds, see e.g. [3, 21].

However, the power of SO tgds comes at a cost: many reasoning tasks become undecidable. For example, even logical equivalence between s-t SO tgds is undecidable [15]. For a survey see e.g. [27].

Yet, there is a middle ground between tgds and SO tgds: nested tgds [16]. Nested tgds were introduced as part of IBM's Clio system [19], which is now part of the InfoSphere BigInsights suite. It has recently been shown that nested tgds have a number of advantages in terms of decidability of reasoning tasks, in particular that equivalence of s-t nested tgds is decidable [22]. Let us now return to our running example. If our schema in addition contains a relation Dep representing departments, then we can express our dependency as the following nested tgd:

$$\forall d\ \mathsf{Dep}(d) \rightarrow \exists dm\,[\forall e\,\mathsf{Emp}(e,d) \rightarrow \mathsf{Mgr}(e,dm)]$$

Looking at our nested tgd in its Skolemized form and normalized to a single implication, again the distinctive feature compared to tgds is the much more flexible use of terms based on function symbols:

$$\exists f_{\mathsf{dm}}\,\forall e,d\ \ \mathsf{Dep}(d) \wedge \mathsf{Emp}(e,d) \rightarrow \mathsf{Mgr}(e, f_{\mathsf{dm}}(d))$$

We have seen that nested tgds are one way to avoid the complexity of SO tgds, but still be able to model interesting domains. They have however one major restriction: they can only model hierarchical relationships (i.e., the argument lists of Skolem functions must form a tree). However, there are natural relationships that can not be captured by nested tgds. Let us extend our example as follows: for every employee, we want to create an employee ID. We can express this as an SO tgd:

$$\exists f_{\mathsf{eid}}, f_{\mathsf{dm}}\,\forall e,d\ \ \mathsf{Emp}(e,d) \rightarrow \mathsf{Mgr}(f_{eid}(e), f_{dm}(d))$$

Nested tgds are not able to express this dependency. So how can we gain a more fine-grained understanding of, in general, not hierarchical relationships without resorting to SO tgds? To answer this question, let us look at a well-known formalism from logic which can help us in that regard. In logic, Henkin quantifiers [29, 8] are a tool to gain a fine-grained control over the way function symbols can occur in the Skolemized form of formulas. Let us write our dependency using a (so-called standard) Henkin quantifier:

$$\begin{pmatrix} \forall d\,\exists dm \\ \forall e\,\exists eid \end{pmatrix} \mathsf{Emp}(e,d) \rightarrow \mathsf{Mgr}(eid, dm)$$

where the quantifier prefix means that the existential variable dm only depends on the department d, and the existential variable eid only depends on the employee e. We shall call such Henkin-quantified rules "Henkin tgds".

Altogether, we have described four "families" of tgds so far: tgds as the least expressive, SO tgds as the most expressive and nested tgds and Henkin tgds in between. In Figure 1, we summarize all the classes of tgds relevant in this paper. We will give formal preliminaries

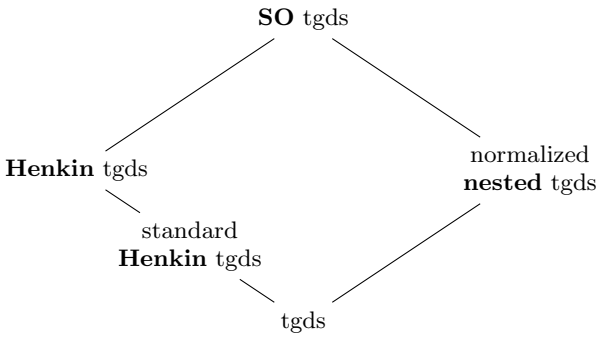


Figure 1: *Hasse diagram of syntactical inclusion between dependency classes in their Skolemized form. An edge denotes that every dependency of the lower class is also a dependency of the upper class.*

for tgds, SO tgds and nested tgds in Section 2. In Section 3 we will formally introduce Henkin tgds as well as discuss normalization of nested tgds. Immediately, this syntactical inclusion diagram raises the question:

- What is the relative expressive power of the given classes? That is, when can we represent tgds from one class as equivalent tgds in another class?

We shall fully answer this question for all classes in Figure 1. It will turn out that the semantical inclusion diagram (given later in Section 4) looks a bit different than the syntactical inclusion diagram in Figure 1.

A central task for database systems is query answering. Given a database, a set of dependencies, and typically a conjunctive query, the task of query answering is to compute the set of certain answers to that conjunctive query. However, query answering for tgds is in general undecidable [20]. Hence, numerous criteria for ensuring decidability have been introduced throughout the last few years. We give an overview by looking at the three major "families" (illustrated in Figure 2).

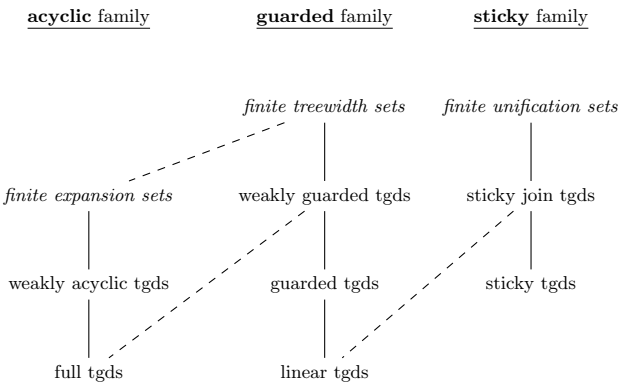


Figure 2: *Hasse diagram based on [24] showing the three major families for decidable query answering of tgds. Solid edges denote inclusions inside the families, dashed edges denote inclusions between the families.*

The best-known family of criteria for ensuring decidable query answering is that of *acyclic*, *weakly acyclic* and *full* tgds [13]. A second family of criteria centers around *guarded* tgds [9], its subset *linear* tgds and its generalization *weakly guarded* tgds [9]. A third family is that of *sticky* tgds [10, 12] The *sticky* and *guarded* (in particular *linear*) based families were combined in the criterion called *sticky join* tgds [11]. A semantic categorization of the different decidable fragments of tgds was done in [5, 6]. The *acyclicity* family is member of tgds with *finite expansion sets* (fes). The *guarded* family is member of tgds with *finite treewidth sets* (fts), a superclass of *fes*. The *sticky* family is member of tgds with *finite unification sets* (fus).

The complex landscape of decidability criteria thus raises the question:

- Where is the decidability/undecidability border for nested, Henkin, and SO tgds, in particular under the three "families" (weakly-)acyclic, guarded and sticky?

Apart from query answering, a second central problem with any logical formalism is model checking. Given a database and a set of dependencies, the task of model checking is to answer whether or not the database satisfies all dependencies.

For tgds, query and combined complexity was shown to be Π_2P-complete [2] (cf. [4]) while – as tgds are first-order formulas – data complexity is in AC_0. For SO tgds, data complexity was shown to be NP-complete [14] and query and combined complexity is known to be NEXPTIME-complete [25]. From these results, we can already derive some bounds for other formalisms, since lower bounds propagate along generalizations and upper bounds propagate along specialization. So the immediate question is:

- What is the precise (data/query/combined) complexity of model checking for nested tgds and Henkin tgds?

Organization and main results. In Section 2, we give preliminaries and in particular recall the classical formalisms of tgds, SO tgds and nested tgds. In Section 3, we introduce Henkin tgds, and discuss normalization of nested tgds. A conclusion and outlook to future work are given in Section 7. Our main results are detailed in Sections 4-6, namely:

- *Expressive Power*. In Section 4, we compare the expressive power of the dependency classes given in Figure 1. Interestingly, we obtain that nested tgds can always be transformed into a logically equivalent set of Henkin tgds. In contrast, we show that a number of inclusions are proper for other classes, in total obtaining a complete picture of the relative expressive power of all classes in Figure 1.
- *Query Answering*. In Section 5, we consider query answering for the dependency classes given in Figure 1. In particular, we show that even when assuming our dependencies to be *both* guarded and sticky, atomic query answering is undecidable for standard Henkin tgds and nested tgds, the two lowest extensions of tgds given in Figure 1. For standard Henkin tgds, undecidability holds even for linear dependencies. In contrast, weak acyclicity guarantees decidability of query answering even for SO tgds. Likewise, imposing a further restriction on linear Henkin tgds leads to decidability. In total, for all discussed dependency classes, we draw a clear border of decidability/undecidability in Figure 2.
- *Model Checking*. In Section 6, we consider the model checking problem. We show that Henkin tgds are NEXPTIME-complete in query and combined complexity and NP-complete in data complexity. Hardness holds even for standard Henkin tgds. We also show that nested tgds are PSPACE-complete in query and combined complexity (while data complexity is known to be in AC_0). Thus we complete the picture of complexity for all the dependency classes in Figure 1.

2. Preliminaries

We assume basic familiarity with logic (predicate logic, Skolemization), complexity and database theory [1].

Schemas, instances, and homomorphisms. A *schema* $\mathcal{R}$ is a finite sequence $\langle R_1, \ldots, R_k \rangle$ of relation symbols, where each R_i has a fixed arity. An *instance* I over $\mathcal{R}$, or an $\mathcal{R}$-*instance*, is a sequence $(R_1^I, \ldots, R_k^I)$, where each R_i^I is a finite relation of the same arity as R_i. We will often use R_i to denote both the relation symbol and the relation R_i^I that instantiates it. A *fact* of an instance I (over $\mathcal{R}$) is an expression $R_i^I(v_1, \ldots, v_m)$ (or simply $R_i(v_1, \ldots, v_m)$), where R_i is a relation symbol of $\mathcal{R}$ and $(v_1, \ldots, v_m) \in R_i^I$.

Sometimes it is beneficial to split a schema into two disjoint parts: Let $\mathcal{S}$ and $\mathcal{T}$ be two schemas with no relation symbols in common. We refer to $\mathcal{S}$ as the *source schema*, and $\mathcal{T}$ as the *target schema*. Similarly, we refer to $\mathcal{S}$-instances as *source instances*, and $\mathcal{T}$-instances as *target instances*. We assume the presence of two kinds of values, namely *constants* and *(labeled) nulls*. We also assume that the active domains of source instances consists of constants; the active domains of target instances may consist of constants and nulls.

Let J_1 and J_2 be two instances. A function h is a *homomorphism* from J_1 to J_2 if the following hold: (i) for every constant c, we have that $h(c) = c$; and (ii) for every relation symbol R in $\mathcal{R}$ and every tuple $(a_1, \ldots, a_n) \in R^{J_1}$, we have that $(h(a_1), \ldots, h(a_n)) \in R^{J_2}$. We use the notation $J_1 \to J_2$ to denote that there is a homomorphism from J_1 to J_2. We say that J_1 is *homomorphically equivalent* to J_2, written $J_1 \leftrightarrow J_2$, if $J_1 \to J_2$ and $J_2 \to J_1$. A minimal subinstance of J that is homomorphically equivalent to J is called a *core* of J. All cores of an instance are isomorphic [18]. Hence, it is justified to speak of *the* core of an instance J, denoted $core(J)$.

Tgds. A *tuple-generating dependency* (in short, *tgd*) σ is a first-order sentence of the form

$$\forall \bar{x}(\varphi(\bar{x}) \to \exists \bar{y} \psi(\bar{x}, \bar{y}))$$

where $\varphi(\bar{x})$ is a conjunction of atoms, each variable in $\bar{x}$ occurs in at least one atom in $\varphi(\bar{x})$, and $\psi(\bar{x}, \bar{y})$ is a conjunction of atoms with variables in $\bar{x}$ and $\bar{y}$. If $\bar{y}$ is empty, then σ is called *full*. If φ consists solely of atoms

over a schema $\mathcal{S}$ and ψ consists solely of atoms over a schema $\mathcal{T}$, we call σ a *source-to-target* (s-t) tgd. For simplicity, we will often suppress writing the universal quantifiers $\forall\bar{x}$ in formulas of the above form.

SO tgds. *Second-order tgds*, or *SO tgds*, were introduced in [14], where it was shown that SO tgds are needed to specify the composition of an arbitrary number of schema mappings based on s-t tgds. Before we formally define SO tgds, we need to define *terms*. Given collections $\bar{x}$ of variables and $\bar{f}$ of function symbols, a *term (based on $\bar{x}$ and $\bar{f}$)* is defined recursively as follows: (1) Every variable in $\bar{x}$ is a term; (2) If f is a k-ary function symbol in $\bar{f}$ and $t_1, \ldots, t_k$ are terms, then $f(t_1, \ldots, t_k)$ is a term.

A *second-order tuple-generating dependency (SO tgd)* is a formula σ of the form:

$$\exists\bar{f}((\forall\bar{x}_1(\varphi_1 \rightarrow \psi_1)) \wedge ... \wedge (\forall\bar{x}_n(\varphi_n \rightarrow \psi_n)))$$

where (1) Each member of $\bar{f}$ is a function symbol. (2) Each φ_i is a conjunction of (i) atoms $S(y_1, ..., y_k)$, where S is a k-ary relation symbol and $y_1, \ldots, y_k$ are variables in $\bar{x}_i$, not necessarily distinct, and (ii) equalities of the form $t = t'$ where t and t' are terms based on $\bar{x}_i$ and $\bar{f}$. (3) Each ψ_i is a conjunction of atoms $T(t_1, ..., t_l)$, where T is an l-ary relation symbol and $t_1, \ldots, t_l$ are terms based on $\bar{x}_i$ and $\bar{f}$. (4) Each variable in $\bar{x}_i$ appears in some atomic formula of φ_i.

Let $\mathcal{S}$ be a source schema and $\mathcal{T}$ a target schema. If all φ_i consist solely of atoms over $\mathcal{S}$ and all ψ_i consist solely of atoms over $\mathcal{T}$, we say that σ is *source-to-target*. Note that in the original definition [14], SO tgds are defined to be always source-to-target. In this paper, we do not make that assumption and use the name "SO tgds" to refer to not necessarily s-t dependencies, while we specifically speak of "s-t SO tgds" when it is necessary to refer to the source-to-target case. Let us give an example of an (s-t) SO tgd.

$$\exists f_{\mathsf{mgr}}(\forall e(\mathsf{Emp}(e) \rightarrow \mathsf{Mgr}(e, f_{\mathsf{mgr}}(e))) \ \wedge \\ \forall e(\mathsf{Emp}(e) \wedge (e = f_{\mathsf{mgr}}(e)) \rightarrow \mathsf{SelfMgr}(e)))$$

The formula expresses the property that every employee has a manager, and if an employee is the manager of himself/herself, then he/she is a self-manager [14].

Note that SO tgds allow for nested terms and for equalities between terms. A nested term is a functional term which contains a functional term as an argument. A *plain SO tgd* is an SO tgd that contains no nested terms and no equalities. For example, the preceding SO tgd is not plain, while the following SO tgd (expressing that if two employees are in some relationship – e.g., working in the same project – also their manager are in some relationship) is plain

$$\exists f_{\mathsf{mgr}}\, \forall e_1, e_2\, (\mathsf{Emps}(e_1, e_2) \rightarrow \mathsf{Mgrs}(f_{\mathsf{mgr}}(e_1), f_{\mathsf{mgr}}(e_2)))$$

The properties of plain SO tgds were recently investigated in [4]. In what follows, we will often suppress writing the existential second-order quantifiers and the universal first-order quantifiers in front of SO tgds.

Nested tgds. Fix a partition of the set of first-order variables into two disjoint infinite sets X and Y. A *nested tgd* [14, 28] is a first-order sentence that can be generated by the following recursive definition:

$$\chi ::= \alpha \mid \forall\bar{x}\,(\beta_1 \wedge \ldots \wedge \beta_k \rightarrow \exists\bar{y}\,(\chi_1 \wedge \ldots \wedge \chi_\ell))$$

where each $x_i \in X$, each $y_i \in Y$, α is a relational atom, and each β_j is a relational atom containing only variables from X, such that each x_i occurs in some β_j. As an example, the following formula is a nested tgd:

$$\forall d\ \mathsf{Dep}(d) \rightarrow \exists dm\, \mathsf{Dep}'(d, dm) \wedge \\ [\forall e\, \mathsf{Emp}(e, d) \rightarrow \mathsf{Mgr}(e, d, dm)]$$

Note that the choice of brackets (round or square) and the weight of the font (normal or bold) is for emphasizing the nesting structure only and has no syntactical significance.

A nested tgd σ contains a number of *parts* σ_i. Informally, σ_i is an implicational formula that contains conjunctions of atoms on both sides (but no nested implications). As an example, the parts of the preceding nested tgd are

- $\forall d\ \mathsf{Dep}(d) \rightarrow \exists dm\, \mathsf{Dep}'(d, dm)$
- $\forall e\, \mathsf{Emp}(e, d) \rightarrow \mathsf{Mgr}(e, d, dm)$

In our examples, we refer to parts of nested tgds using labels. The way of inline labeling of parts which we use throughout this paper is illustrated below by a nested tgd τ with three parts τ_1, τ_2, τ_3:

$$\forall d\, \mathsf{Dep}(d) \rightarrow \exists d'\, \mathsf{Dep}'(d') \wedge \qquad (\tau_1)$$
$$[\forall g\, \mathsf{Grp}(d, g) \rightarrow \exists g' \mathsf{Grp}'(d', g') \wedge \qquad (\tau_2)$$
$$[\forall e\, \mathsf{Emp}(d, g, e) \rightarrow \mathsf{Emp}'(d', g', e)]] \qquad (\tau_3)$$

It is often convenient to consider *Skolemized nested tgds*, in which every existential variable y is replaced by the Skolem term $f(\bar{x})$ where f is a fresh function symbol and $\bar{x}$ is the vector of universally quantified variables in the part σ_i in which $\exists y$ occurs, and in the ancestors of σ_i. Note that we assume existential variables in different parts to be renamed apart. The Skolemized version of nested tgd τ above has the following form:

$$\exists \boldsymbol{f_{\mathsf{d}}, f_{\mathsf{g}}}\, \mathsf{Dep}(d) \rightarrow \mathsf{Dep}'(f_{\mathsf{d}}(d)) \wedge \qquad (\tau_1)$$
$$[\mathsf{Grp}(d, g) \rightarrow \mathsf{Grp}'(f_{\mathsf{d}}(d), f_{\mathsf{g}}(d, g)) \wedge \qquad (\tau_2)$$
$$[\mathsf{Emp}(d, g, e) \rightarrow \mathsf{Emp}'(f_{\mathsf{d}}(d), f_{\mathsf{g}}(d, g), e)]] \qquad (\tau_3)$$

Note that syntactically, a Skolemized nested tgd is not necessarily an SO tgd, as it may contain nested implications. However, given a nested tgd it is easy to create a logically equivalent SO tgd where the number of parts of the SO tgd is the same as the number of parts of the nested tgd. Thus one can, informally, consider nested tgds as a subclass of SO tgds. We will discuss this issue in detail in Section 3.

Queries. A conjunctive query (CQ) q over a schema $\mathcal{R}$ with free variables $\bar{y}$ is a logical formula of the form

$$\exists\bar{x}\,(A_1 \wedge \ldots \wedge A_n)$$

where each A_i is a relational atom over $\mathcal{R}$ with variables from $\bar{x} \cup \bar{y}$, where $\bar{x}$ and $\bar{y}$ are disjoint. If $\bar{y}$ is empty, we call q a Boolean conjunctive query. Given a database instance I, the set of answers $q(I)$ to a query q is the set $\{\bar{y} \mid I \vDash q\}$. Given a set Σ of dependencies and a database instance I, the *certain answers* to a conjunctive query q w.r.t. I are those which are answers over all database instances $\{I' \mid I \subseteq I' \wedge I' \vDash \Sigma\}$. Given a set of source-to-target dependencies, queries are usually posed against the target schema only.

The *query answering* problem has as input a database instance I, a set of dependencies Σ, a query q and a tuple t. The question is whether t is contained in all certain answers to q w.r.t. I and Σ. The problem is called *atomic query answering* if query q consists of a single atom.

Equivalence. Two sets of dependencies Σ_1 and Σ_2 are called *logically equivalent*, written $\Sigma_1 \equiv \Sigma_2$ or $\Sigma_1 \equiv_{\mathsf{log}} \Sigma_2$, if they have the same models. That is, $I \vDash \Sigma_1$ holds if and only if $I \vDash \Sigma_2$ holds. They are called *CQ-equivalent*, written $\Sigma_1 \equiv_{\mathsf{CQ}} \Sigma_2$ if they have the same certain answers over all conjunctive queries q and database instances I. Clearly, logical equivalence implies CQ-equivalence, but the converse does not necessarily hold. Note that logical and CQ-equivalence always consider dependencies over the same schema.

3. Henkin and Normalized Nested tgds

So far we have recalled tgds (the bottom part of Figure 1) and SO tgds (the top part of Figure 1), and we have started giving preliminaries for nested tgds. In this section, our main goal is to introduce Henkin tgds and their various flavors. After that, based on the definition of nested tgds given in the preliminaries, we will discuss how to treat nested tgds as a subclass of SO tgds.

3.1 Henkin tgds

We first introduce Henkin quantifiers and then use them for building dependencies.

A *Henkin quantifier* (cf. [8, 29]) is given by

- a set of first-order quantifiers
- a strict partial order between these quantifiers (i.e., an irreflexive, transitive relation) [3]

The semantics of Henkin quantifiers is given by their Skolemization, that is, the Skolem term of an existential variable contains all universally quantified variables that are preceding the existential variable in the given partial order. For example, recall the following plain SO tgd τ from the introduction:

$$\exists f_{\mathsf{eid}}, f_{\mathsf{dm}}\, \forall e, d\ \ \mathsf{Emp}(e,d) \rightarrow \mathsf{Mgr}(f_{eid}(e), f_{dm}(d))$$

This dependency could be obtained through Skolemization of the formula

$$Q\ \mathsf{Emp}(e,d) \rightarrow \mathsf{Mgr}(eid, dm)$$

under a Henkin quantifier Q given by the partial order $\prec$ with $\forall e \prec \exists eid$ and $\forall d \prec \exists dm$.

The first important observation is that it is only relevant which universal quantifiers come before an existential quantifier. This is intuitively clear when inspecting the Skolemization of a formula, as the arguments of the Skolem functions for an existential quantifier are purely determined by the scope of the universal quantifiers. This relevant part of the order is called the *essential order* [29].

A Henkin quantifier is called *standard* if

- the strict partial order consists of (disjoint) chains (i.e., it is the disjoint union of a set of linear orders)
- the strict partial order is already an essential order (i.e., every chain consists of universal quantifiers followed by existential quantifiers) [4]

Standard Henkin quantifiers are usually denoted as

$$\begin{pmatrix} \forall \bar{x}_1 & \exists \bar{y}_1 \\ \cdots & \cdots \\ \forall \bar{x}_n & \exists \bar{y}_n \end{pmatrix}$$

where $\bar{x}_1$ to $\bar{x}_n$ and $\bar{y}_1$ to $\bar{y}_n$ are vectors of variables, and all variables are distinct. In this notation, each of the rows of the quantifier represents a chain of the strict partial order. For example, our dependency τ:

$$\mathsf{Emp}(e,d) \rightarrow \mathsf{Mgr}(f_{\mathsf{eid}}(e), f_{\mathsf{dm}}(d))$$

can be produced through Skolemization of the formula

$$\begin{pmatrix} \forall d\, \exists dm \\ \forall e\, \exists eid \end{pmatrix} \mathsf{Emp}(e,d) \rightarrow \mathsf{Mgr}(eid, dm)$$

In contrast, note that the following expression σ does *not* contain a standard Henkin quantifier

$$\begin{pmatrix} \forall x_1 & \forall x_2 & \exists y_1 \\ \forall x_2 & \forall x_3 & \exists y_2 \\ \forall x_3 & \forall x_1 & \exists y_3 \end{pmatrix} \varphi(x_1, x_2, x_3, y_1, y_2, y_3)$$

This is the case, since the chains are not disjoint. Note that, in first-order logic (with equality), every positive occurrence of a Henkin quantifier can be expressed by a standard Henkin quantifier [8]. For example, σ can be expressed as:

$$\begin{pmatrix} \forall x_1 & \forall x_2 & \exists y_1 \\ \forall x'_2 & \forall x_3 & \exists y_2 \\ \forall x'_3 & \forall x'_1 & \exists y_3 \end{pmatrix} \ [x_1 = x'_1 \wedge x_2 = x'_2 \wedge x_3 = x'_3] \rightarrow \varphi(x_1, x_2, x_3, y_1, y_2, y_3)$$

That is, all occurrences of a variable are given unique names and associated using equalities. However, this is not a technique we can use for defining a lightweight subclass of plain SO tgds, since plain SO tgds do not allow equalities in the antecedent. Thus for our purposes, whether we have standard Henkin quantifiers or Henkin quantifiers makes a difference.

We are now ready to give a formal definition of the family of Henkin tgds.

Definition 3.1 *Let $\mathcal{C}$ be a class of Henkin quantifiers, let $\mathcal{Q}$ be a Henkin quantifier from $\mathcal{C}$. Then a $\mathcal{C}$-Henkin tgd is a formula of the form*

$$\mathcal{Q}\ (\varphi(\bar{x}) \rightarrow \psi(\bar{x}, \bar{y}))$$

where $\bar{x}$ consists of universally quantified variables in $\mathcal{Q}$ and $\bar{y}$ consists of existentially quantified variables in $\mathcal{Q}$.

In particular, if $\mathcal{C}$ is the class of all Henkin quantifiers, we simply speak of *Henkin tgds* and if $\mathcal{C}$ is the class of standard Henkin quantifiers, we speak of *standard Henkin tgds*. Furthermore, we will also use the class where the partial order of the Henkin quantifier is tree-structured, hence *tree Henkin tgds* (or more formally,

[3] Sometimes also a non-strict partial order is assumed. Here we follow [29] and assume a strict partial order.

[4] We make the following assumption here which deviates from the definition of [29], to ensure that normal FO quantifiers are standard Henkin quantifiers: A chain may end in multiple existential quantifiers.

every connected component of the graph of the partial order is a tree). From here on, we will always consider the Skolemized form of dependencies.

3.2 Normalized and Simple Nested tgds

So far, we have described all dependency classes in Figure 1, except for nested tgds. This is for a simple reason: In terms of syntax, a nested tgd is not necessarily an SO tgd, as it may contain nested implications. Yet intuitively, one can undo this nesting by applying well-known equivalences from first-order logic. We will formalize this "normalization" now. Let us first consider the following (Skolemized) nested tgd τ as an example:

$$\exists \boldsymbol{f_d}, \boldsymbol{f_g}\, \mathsf{Dep}(d) \rightarrow \mathsf{Dep}'(f_{\mathsf{d}}(d)) \wedge \quad (\tau_1)$$
$$[\mathsf{Grp}(d,g) \rightarrow \mathsf{Grp}'(f_{\mathsf{d}}(d), f_{\mathsf{g}}(d,g)) \wedge \quad (\tau_2)$$
$$[\mathsf{Emp}(d,g,e) \rightarrow \mathsf{Emp}'(f_{\mathsf{d}}(d), f_{\mathsf{g}}(d,g), e)]] \quad (\tau_3)$$

Intuitively, this nested tgd τ takes a three-level hierarchy of departments, groups and employees, and simply invents identifiers for departments and groups. Let us remove nesting levels one-by-one, starting at the innermost level. By using the fact that $\varphi \rightarrow (\psi \wedge [\varphi_1 \rightarrow \psi_1])$ is equivalent to $[\varphi \rightarrow \psi] \wedge [\varphi \wedge \varphi_1 \rightarrow \psi_1]$ we thus obtain the following two-level nested tgd:

$$\exists \boldsymbol{f_d}, \boldsymbol{f_g}\, \mathsf{Dep}(d) \rightarrow \mathsf{Dep}'(f_{\mathsf{dep}}(d)) \wedge \quad (\tau_1')$$
$$[\mathsf{Grp}(d,g) \rightarrow \mathsf{Grp}'(f_{\mathsf{d}}(d), f_{\mathsf{g}}(d,g))] \wedge \quad (\sigma_2)$$
$$[\mathsf{Grp}(d,g) \wedge \mathsf{Emp}(d,g,e) \rightarrow \mathsf{Emp}'(f_{\mathsf{d}}(d), f_{\mathsf{g}}(d,g), e)] \quad (\sigma_{23})$$

In particular, we have applied the equivalence to τ_1 and τ_2, obtaining σ_2 and σ_{23}. Let us now remove the outermost nesting level. This time, we have two nested dependencies σ_2 and σ_{23}, but the idea of applying the equivalence remains the same:

$$\exists \boldsymbol{f_d}, \boldsymbol{f_g}\, [\mathsf{Dep}(d) \rightarrow \mathsf{Dep}'(f_{\mathsf{dep}}(d))] \wedge \quad (\sigma_1)$$
$$[\mathsf{Dep}(d) \wedge \mathsf{Grp}(d,g) \rightarrow \mathsf{Grp}'(f_{\mathsf{d}}(d), f_{\mathsf{g}}(d,g))] \wedge \quad (\sigma_{12})$$
$$[\mathsf{Dep}(d) \wedge \mathsf{Grp}(d,g) \wedge \mathsf{Emp}(d,g,e) \rightarrow$$
$$\mathsf{Emp}'(f_{\mathsf{d}}(d), f_{\mathsf{g}}(d,g), e)] \quad (\sigma_{123})$$

Thus, we now have a conjunction of three (non-nested) implications, all bound by the same quantifier prefix $\exists f_{\mathsf{d}}, f_{\mathsf{g}}$ or, in other words, a plain SO tgd. In Algorithm 1, we formally describe the transformations we have applied in our example. We thus get the following straightforward definition of normalization.

Algorithm 1: nested-to-so
Input: nested tgd τ in Skolemized form
Output: plain SO tgd σ s.t. $\sigma \equiv_{\mathsf{log}} \tau$

1. Apply the following rewrite step recursively (innermost nesting level to outermost level):

$$\overbrace{\varphi \rightarrow (\psi}^{\text{tgd}} \wedge [\overbrace{\varphi_1 \rightarrow \psi_1}^{\text{plain SO tgd}}] \wedge \ldots \wedge [\overbrace{\varphi_n \rightarrow \psi_n}^{\text{plain SO tgd}}])$$

$$\Downarrow$$

$$\varphi \rightarrow \psi \wedge \bigwedge_{i \in \{1,\ldots,n\}} \underbrace{[\varphi \wedge \varphi_i \rightarrow \psi_i]}_{\text{plain SO tgd}}$$

2. Return σ, which is the result of recursively applying the rewrite rule.

Definition 3.2 *Let τ be a nested tgd. The* normalized *form of τ is the plain SO tgd $\sigma = \mathsf{nested\text{-}to\text{-}so}(\tau)$.*

Using this definition, we speak of *normalized nested tgds*. We now have defined all dependency classes shown in Figure 1. Furthermore, in what follows, it will often be convenient to use the following restriction of SO tgds: We call an SO tgd *simple*, if it contains exactly one SO tgd part. Similarly, we can speak of *simple nested tgds* if a normalized nested tgd consists of only one SO tgd part. For example, our dependency $\sigma = \mathsf{nested\text{-}to\text{-}so}(\tau)$ is not a simple nested tgd, as it contains three SO tgd parts.

4. Expressive Power

In this section, we investigate the relative expressive power of the dependency classes, that is, when for a given dependency in one class, we can find a logically equivalent set of dependencies in other classes. An overview of this semantical relationship can be found in Figure 3.

In the previous section, we have seen that normalized nested tgds allow the quantification of Skolem function symbols over several dependencies, whereas Henkin tgds only allow quantification over each individual dependency. Nevertheless, quite surprisingly, we shall show in this section that nested tgds can always be expressed as a logically equivalent set of Henkin tgds, indicated by the bold (blue) edge in the Hasse diagram. We shall give this result later in this section. The dotted (red) edges in the Hasse diagram indicate that for a given set of dependencies, not even a CQ-equivalent set of dependencies exists. In particular, all dotted edges together imply that

1. absent (non-implied) edges in the Hasse diagram do not exist. That is, standard Henkin and nested tgds are incomparable with regard to expressive power.
2. all edges denote proper containment. That is, for any two distinct classes, there exists a set of dependencies that separates these two classes.

We now proceed to show the separations (dotted or red edges) in Figure 3. We first separate standard Henkin tgds from nested tgds. We start by giving high-level ideas that will play a major role in the following proofs.

Idea 1: Basic Construction. All of our proofs that a set of dependencies $\Sigma \subseteq \mathcal{C}_1$ is not expressible as a CQ-equivalent set $\Pi \subseteq \mathcal{C}_2$ will proceed by contradiction, in particular along the following general lines:

1. assume that a CQ-equivalent set $\Pi \subseteq \mathcal{C}_2$ exists
2. define an instance I such that our dependencies have to create a particular "large" structure using the specific features of $\mathcal{C}_1$
3. show that $\mathcal{C}_2$ needs to "pass" nulls (values stemming from functional terms) to create the large structure (see Idea 2)
4. show that recursion is of little help for creating the large structure (see Idea 3)
5. derive a contradiction $\lhd$

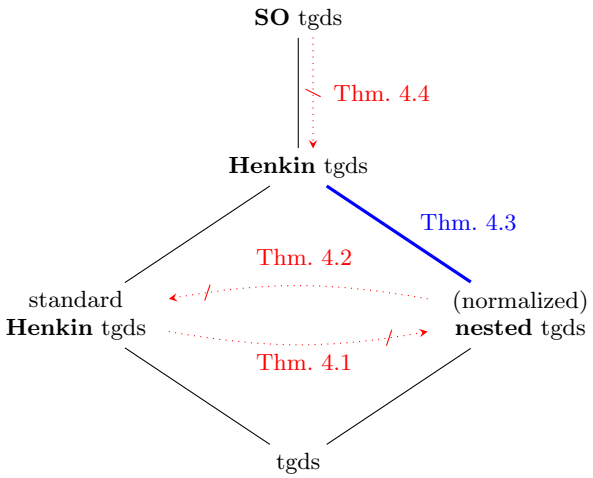


Figure 3: *Hasse diagram of the semantical inclusions between classes. A solid edge denotes that every set of dependencies from the lower class can be expressed as a logically equivalent set of dependencies from the upper class.*

Idea 2: "Passing of Nulls". Every class of dependencies has particular structures of nulls it can generate. As a simple example, nulls generated by a nested tgd have a tree structure, as can be seen in Skolemized form:

$$P(x_1) \wedge Q(x_1, x_2) \rightarrow R(f(x_1), g(x_1, x_2))$$

In this case (i.e., in the case that Skolem terms are at the corresponding positions), we say that the nulls are *directly generated* by our nested tgd. Yet this does not prevent more complex structures to be created by passing of nulls, i.e., dependencies of the form

$$\varphi \wedge R_i(\ldots, x, \ldots) \rightarrow \psi \wedge R_j(\ldots, x, \ldots)$$

where x is bound to a null value that is *passed* from R_i to R_j. Showing that passing of nulls is necessary to create more complex structures is essential in the proofs that follow. ◁

Idea 3: "Recursion Poisoning". Apart from tgds in Σ that exhibit the actual behavior to create our "large" structure, we use dependencies that allow us to add additional atoms into that structure. Assume that our large structure is encoded using facts of the form $R(x_1, x_2, x_3)$. Then we add an additional source of such R facts through the simple copy tgd

$$R'(x_1, x_2, x_3) \rightarrow R(x_1, x_2, x_3)$$

This additional source of R facts allows us to forbid many of the potential recursive dependencies using R, as they would be violated by the "poisoned" facts introduced by the copying tgd. ◁

We are now ready to separate standard Henkin tgds from nested tgds. We start by showing that there are standard Henkin tgds which are not expressible as nested tgds. Note that in Theorem 4.1 (and analogously in the subsequent separation theorems) we actually show a slightly stronger result, namely: even if we restrict standard Henkin tgds to *source-to-target* dependencies and allow *arbitrary* nested tgds, then, in general, standard Henkin tgds cannot be expressed by nested tgds.

Theorem 4.1 *There exists a set Σ of s-t standard Henkin tgds such that there is no set Π of (not necessarily s-t) nested tgds with $\Pi \equiv_{\mathsf{CQ}} \Sigma$.*

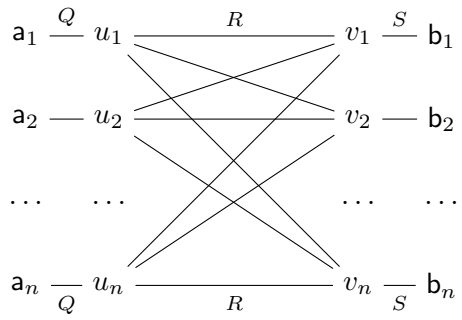
Proof (Sketch) Let Σ consist of dependencies

$$\begin{aligned}
&P(x_1, x_2) \rightarrow Q(x_1, f(x_1)) \wedge \\
&\quad R(f(x_1), g(x_2)) \wedge S(g(x_2), x_2) && (\sigma_1)\\
&Q'(x_1, x_2) \rightarrow Q(x_1, x_2) && (\sigma_2)\\
&R'(x_1, x_2) \rightarrow R(x_1, x_2) && (\sigma_3)\\
&S'(x_1, x_2) \rightarrow S(x_1, x_2) && (\sigma_4)
\end{aligned}$$

In essence, σ_1 is used for encoding the intended behavior, and σ_2 to σ_4 are used for the "recursion poisoning" technique described in Idea 3. Towards a contradiction, assume that there exists a set Π of nested tgds with $\Pi \equiv_{\mathsf{CQ}} \Sigma$. Following Idea 1, we now proceed to constructing our "large" instance I. Let n be greater than the size of Π (the number of atoms contained in Π). We construct the following source instance:

$$I = \{P(\mathsf{a}_i, \mathsf{b}_j) \mid 1 \leq i, j \leq n\}$$

The structure of the intended target instance is sketched below:



Intuitively, Σ produces a complete bipartite graph encoded through nulls u_i resp. v_j in R. On the left and right sides, this bipartite graph is "protected" by constants through Q and S facts. More formally, the constants contained in Q and S prevent the R facts from being mapped by a homomorphism to a single R fact in the core.

Following Idea 2, our next step is to show that "passing of nulls" is required. The high-level reason for why nested tgds cannot generate the structure shown in the above figure without passing of nulls is as follows: In atoms generated by multiple firings of a nested tgd, there is always a functional dependency between two attributes generated by nulls; e.g., in a normalized nested tgd of the form

$$P(x_1) \wedge Q(x_1, x_2) \rightarrow R(f(x_1), g(x_1, x_2))$$

there is a functional dependency from the attribute generated by g to f (if both functional terms contain the same argument list, then there is a functional dependency in both directions). Thus a nested tgd may directly generate only n of the n^2 edges of our complete bipartite graph, and passing of nulls is required to generate the rest.

We may now use the "recursion poisoning" technique according to Idea 3 to derive a contradiction. While we cannot go into the details (full details can be found in

Algorithm 2: nested-to-henkin
Input: nested tgd τ in Skolemized form
Output: Set Σ of Henkin tgds s.t. $\Sigma \equiv_{\mathsf{log}} \tau$

1. Apply the following rewrite step recursively (innermost nesting level to outermost level):

$$\overbrace{\varphi \rightarrow (\psi}^{\text{tgd}} \wedge \overbrace{[\exists \bar{f}_1\, \varphi_1 \rightarrow \psi_1]}^{\text{tree Henkin tgd}} \wedge \ldots \wedge \overbrace{[\exists \bar{f}_n\, \varphi_n \rightarrow \psi_n]}^{\text{tree Henkin tgd}})$$

$$\Downarrow$$

$$\bigwedge_{I \subseteq \{1,\ldots,n\}} \underbrace{[\exists \bar{f} \exists_{i \in I} \bar{f}_i\ \varphi \wedge \bigwedge_{i \in I} \varphi_i \rightarrow \psi \wedge \bigwedge_{i \in I} \psi_i]}_{\text{tree Henkin tgd}}$$

   where $\bar{f}$ is the set of Skolem functions representing existential quantifiers of ψ, and all universally quantified variables of φ_i are assumed to be renamed apart.

2. Return $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ where $\sigma_1 \wedge \ldots \wedge \sigma_k$ is the result of recursively applying the rewriting rule.

the full version) the key idea is to extend source instance I to I' by Q', R' and S' facts that generate (both in Σ and Π) the corresponding Q, R and S facts in the target instance. In Σ, which is source-to-target, these additional target facts are of no concern. Yet in Π, which contains a target tgd τ for the "passing of nulls", the "poisoned" Q, R and S facts cause another triggering of the target tgd τ – thus producing additional facts which are not produced under Σ. This is a contradiction to our assumption that $\Pi \equiv_{\mathsf{CQ}} \Sigma$. □

We now show the reverse direction of the separation, namely that nested tgds are, in general, not expressible as standard Henkin tgds:

Theorem 4.2 *There exists a set Σ of s-t simple nested tgds such that there is no set Π of (not necessarily s-t) standard Henkin tgds with $\Pi \equiv_{\mathsf{CQ}} \Sigma$.*

We now show that the class of nested tgds is semantically contained in the class of tree Henkin tgds. That is, we show the following theorem:

Theorem 4.3 *Let τ be a nested tgd. Then there exists a set $\Sigma = \mathsf{nested\text{-}to\text{-}henkin}(\tau)$ of tree Henkin tgds such that $\Sigma \equiv_{\mathsf{log}} \tau$.*

The algorithm nested-to-henkin for converting a nested tgd into a logically equivalent set of Henkin tgds is given as Algorithm 2. Note that Algorithm 2 has the same general structure as Algorithm 1, which we already discussed in detail in Section 3. Let us now look at an example that shows the essence of the algorithm. Assume that we are given the following nested tgd τ (which we already used to illustrate Algorithm 1):

$$\exists \boldsymbol{f_d}, \boldsymbol{f_g}\, \mathsf{Dep}(d) \rightarrow \mathsf{Dep}'(f_{\mathsf{d}}(d)) \wedge \qquad (\tau_1)$$
$$[\mathsf{Grp}(d,g) \rightarrow \mathsf{Grp}'(f_{\mathsf{d}}(d), f_{\mathsf{g}}(d,g)) \wedge \qquad (\tau_2)$$
$$[\mathsf{Emp}(d,g,e) \rightarrow \mathsf{Emp}'(f_{\mathsf{d}}(d), f_{\mathsf{g}}(d,g), e)]] \qquad (\tau_3)$$

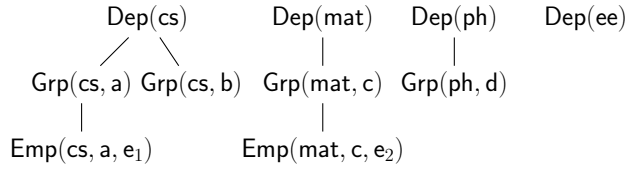
Recall that the nested tgd τ takes a three-level hierarchy of departments, groups and employees, and simply invents identifiers for departments and groups. The basic principle of our algorithm nested-to-henkin is to remove nesting levels one-by-one, starting at the innermost level. Following the rewrite step illustrated in Algorithm 2, we thus produce the following two-level nested dependency:

$$\exists \boldsymbol{f_d}\, \mathsf{Dep}(d) \rightarrow \mathsf{Dep}'(f_{\mathsf{dep}}(d)) \wedge \qquad (\tau_1')$$
$$[\exists \boldsymbol{f_g}\, \mathsf{Grp}(d,g) \rightarrow \mathsf{Grp}'(f_{\mathsf{d}}(d), f_{\mathsf{g}}(d,g))] \wedge \qquad (\sigma_2)$$
$$[\exists \boldsymbol{f_g}\, \mathsf{Grp}(d,g) \wedge \mathsf{Emp}(d,g,e) \rightarrow$$
$$\mathsf{Grp}'(f_{\mathsf{d}}(d), f_{\mathsf{g}}(d,g)) \wedge \mathsf{Emp}'(f_{\mathsf{d}}(d), f_{\mathsf{g}}(d,g), e)] \qquad (\sigma_{23})$$

The critical part of this step is that we have to show that logical equivalence is thus preserved. While the most obvious effect of the rewriting step is the growing number of atoms, the interesting part for logical equivalence is the "splitting" of the quantifiers: originally, we had just one quantifier $\exists f_{\mathsf{d}}$ and now we have two independent ones. Now let us apply the final rewrite step to observe something interesting:

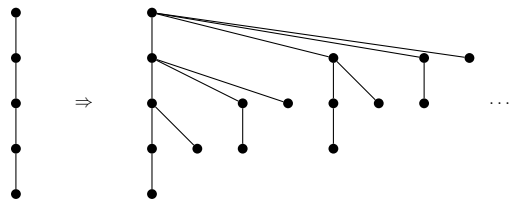
$$\exists \boldsymbol{f_d}, \boldsymbol{f_g}\, \mathsf{Dep}(d) \rightarrow \mathsf{Dep}'(f_{\mathsf{d}}(d)) \qquad (\sigma_1)$$
$$\exists \boldsymbol{f_d}, \boldsymbol{f_g}\, \mathsf{Dep}(d) \wedge \mathsf{Grp}(d,g) \rightarrow$$
$$\mathsf{Dep}'(f_{\mathsf{d}}(d)) \wedge \mathsf{Grp}'(f_{\mathsf{d}}(d), f_{\mathsf{g}}(d,g)) \qquad (\sigma_{12})$$
$$\exists \boldsymbol{f_d}, \boldsymbol{f_g}\, \mathsf{Dep}(d) \wedge \mathsf{Grp}(d,g) \wedge \mathsf{Emp}(d,g,e) \rightarrow$$
$$\mathsf{Dep}'(f_{\mathsf{d}}(d)) \wedge \mathsf{Grp}'(f_{\mathsf{d}}(d), f_{\mathsf{g}}(d,g)) \wedge$$
$$\mathsf{Emp}'(f_{\mathsf{d}}(d), f_{\mathsf{g}}(d,g), e) \qquad (\sigma_{13})$$
$$\exists \boldsymbol{f_d}, \boldsymbol{f_g}\, \mathsf{Dep}(d) \wedge \mathsf{Grp}(d,g) \wedge \mathsf{Emp}(d,g,e) \wedge \mathsf{Grp}(d,g^\star) \rightarrow$$
$$\mathsf{Dep}'(f_{\mathsf{d}}(d)) \wedge \mathsf{Grp}'(f_{\mathsf{d}}(d), f_{\mathsf{g}}(d,g)) \wedge$$
$$\mathsf{Emp}'(f_{\mathsf{d}}(d), f_{\mathsf{g}}(d,g), e) \wedge \mathsf{Grp}'(f_{\mathsf{d}}(d), f_{\mathsf{g}}(d,g^\star)) \qquad (\sigma_{123})$$

Indeed, the preceding four rules are Henkin tgds. Yet, while we previously had three dependencies, we now have four. The first three of them are verbose but relatively straightforward (simply containing the antecedents and conclusions of τ); the fourth dependency σ_{123} is more complex. To illustrate why σ_{123} is needed, let us look at a specific database instance. The essential parts of this instance are sketched below in hierarchical fashion:



In this example, we see the departments of computer science (cs), mathematics (mat), physics (ph) and electrical engineering (ee). For departments mat, ph and ee, it can be checked that the set Σ' of Henkin tgds is sufficient, but for departments cs there is a problem: Consider an instance I containing all previously given facts with constant cs plus additional facts $\mathsf{Grp}'(n_{\mathsf{cs}}, n_{\mathsf{a}})$ and $\mathsf{Grp}'(n^\star_{\mathsf{cs}}, n_{\mathsf{b}})$, where all arguments are nulls. Notice that in the first atom, we have n_{cs} as the identifier for department cs, and a different identifier $n^\star_{\mathsf{cs}}$ in the second atom. Indeed, we have $I \models \Sigma'$ (since we can assign different values to $f_{\mathsf{d}}(\mathsf{cs})$ in σ_{12} and σ_{13}, as the function symbol is separately quantified in σ_{12} and σ_{13}), but $I \not\models \tau$ (since we can assign only one value to $f_{\mathsf{d}}(\mathsf{cs})$). However, once we add σ_{123}, we have that $\Sigma \equiv_{\mathsf{log}} \tau$.

Let us now make an observation concerning the algorithm nested-to-henkin: In a single step, it may produce exponentially many dependencies (notice that we consider all subsets of the index set I in Algorithm 2). Hence, in total, the algorithm may produce non-elementary (in the nesting depth and the number of parts) many Henkin tgds. While this behavior was not so apparent in our example, let us sketch what happens to a five-level nested tgd:



Intuitively, what we see sketched above on the left side is a five-level nested tgd (where each dot denotes a tgd part). On the right side, we see the tree structure covered by the largest Henkin tgd generated by Algorithm 2 (similar to how σ_{123} was the largest Henkin tgd generated by Algorithm 2 for our three-level nested tgd).

Altogether, the algorithm connects nested tgds, Henkin tgds and SO tgds in an interesting way: Obviously, nested tgds have a particular power, namely nesting of implications. SO tgds may emulate this power with only linear blow-up by using one of its powerful features, namely quantifier scope over entire conjunctions of implications (recall the normalization algorithm nested-to-so). Henkin tgds do not have this powerful quantifier scope. Yet, while potentially incurring non-elementary blow-up, they can emulate the power of nested tgds by providing a separate dependency for each possible tree structure of nulls. As future work, it would be interesting to see whether this blowup is indeed unavoidable.

Finally, we show the last missing part for completing Figure 3, namely that there are SO tgds which are not expressible as Henkin tgds:

Theorem 4.4 *There exists an s-t simple plain SO tgd σ such that there is no set Π of (not necessarily s-t) Henkin tgds for which $\Pi \equiv_{\mathsf{CQ}} \sigma$ holds.*

Getting back to the big picture, and in particular Figure 3, notice that the separation results shown in this section also imply separation results for all other edges in Figure 3 that are not covered by explicit results. In particular, since Theorems 4.1 and 4.2 separate standard Henkin tgds and normalized nested tgds from each other, they also separate these two classes from their subclass (tgds) and their superclass (Henkin tgds). So indeed, Figure 3 presents a complete picture of the expressive power of our dependency classes.

5. Query Answering

In this section, we study the problem of query answering. It is structured as follows. First, we show our undecidability results by introducing the problem we are going to reduce from, namely Post's Correspondence Problem. We then give the main ideas of the undecidability proof for linear standard Henkin tgds and show that undecidability also holds for simple nested tgds that are both sticky and guarded. Finally we proceed to decidability. Namely, we discuss that decidability holds in case of weak acyclicity and show that in a limited setting, decidability is also achievable for linear Henkin tgds.

Before we go into the details, let us informally describe what sticky [10] and guarded [9] dependencies are. Note that allowing plain SO tgds rather than ordinary tgds has no effect on the definition of these restrictions.

- a plain SO tgd is called *guarded* if there exists an atom G in the antecedent that contains all variables occurring in the antecedent. That is, assuming φ contains only variables from $\bar{x}$, it is of the form
$$G(\bar{x}) \wedge \varphi(\bar{x}) \rightarrow \psi(\bar{x})$$
- a set of plain SO tgds is called *sticky* if, intuitively speaking, whenever a variable x is joined over, i.e.,
$$R_i(x, \bar{y}_1) \wedge R_j(x, \bar{y}_2) \rightarrow R_k(\bar{z})$$
it must be contained in all conclusion atoms (i.e., above $x \in \bar{z}$). In addition, x may also "never get lost again", that is, any dependency having R_k in the antecedent must propagate a position where x occurs into all conclusion atoms.

Post's Correspondence Problem (PCP). Post's Correspondence Problem is a classical undecidable problem [26]. It is concerned with strings that can be formed by concatenating certain given words over an alphabet. W.l.o.g., let our alphabet be given by the integers $1, \ldots, k$. An instance of PCP is given by the following two parts:

- the alphabet $A = \{1, \ldots, k\}$
- pairs of words over A: $(w_1^1, w_1^2), \ldots, (w_n^1, w_n^2)$

The question of PCP is whether there exists a non-empty sequence $i_1, \ldots, i_\ell$ of indices s.t.
$$w_{i_1}^1 \cdot w_{i_2}^1 \cdot \ldots \cdot w_{i_\ell}^1 = w_{i_1}^2 \cdot w_{i_2}^2 \cdot \ldots \cdot w_{i_\ell}^2$$

In what follows, let $w_{i,j}^s$ denote the j^{th} character of w_i^s. That is, if the j^{th} character of w_i^s is c, then $w_{i,j}^s = c$.

Idea 1: Basic construction. For representing potential solutions to PCP, we represent two types of information:

- sequence of words selected in the solution
- string obtained by concatenating the selected words

We represent both in a standard way using unary function symbols: for instance, a string $(3, 7, 4, \ldots)$ is represented as a term $f_3(f_7(f_4(\ldots)))$. Selected pairs of words are treated similarly: If e.g. first w_5^1 and w_5^2 were selected, followed by w_8^1 and w_8^2, we represent this as $g_5(g_8(\ldots))$.

The key way to use these data structures to obtain the desired result is as follows. We use two "branches" to represent the first and second string of the PCP, marked by 1 resp. 2 in the first argument of atoms with predicate symbol R. An illustration can be found in Figure 4.

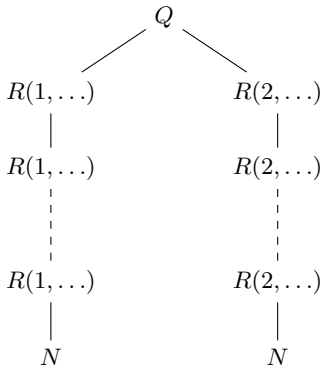


Figure 4: *Intended structure of an instance.*

Both branches start with the empty string and the empty sequence of selected words at the N facts. They then successively select pairs of words and record these selections through R facts. If in the end, both branches produce the same word through the same sequence of selected words, a Q fact is produced. ◁

Idea 2: Representing constants. For our encoding, we need a number of constants for representing e.g. branches, symbols of the alphabet and indexes for words. For representing such constants, the following two properties must be guaranteed:

- symbols that represent the same constant must have the same interpretation
- the sticky property may not be violated

A simple solution to the first issue is to represent the set of constants $\{0, \ldots, d\}$ as a relation N of arity d, where the i^{th} argument represents constant i. This is exactly what the N facts depicted in Figure 4 are used for. The specific solution for also guaranteeing stickiness is as follows. Whenever a fact is introduced, we add a vector of variables representing constants to it:

$$N(\bar{x}) \rightarrow R(\ldots, \bar{x})$$

Whenever a dependency generates a fact, it also includes that vector $\bar{x}$ in that fact. Finally, when the two branches are joined, the join also includes the vector $\bar{x}$ and propagates it to Q. ◁

Idea 3: Applying functions. There is a subtle but critical part when using function symbols in conclusions: For each function symbol, there can only be a single dependency with that function symbol in the conclusion. The reason for this requirement is that it guarantees that the dependencies can actually be obtained from a set of Henkin tgds by Skolemization. However, in our set of dependencies we use function symbols to encode strings. In a naive encoding, we thus have many different dependencies where the same alphabet symbol is added to the string. The solution is a "two-phase" approach.

Say that some R atom should have function f_3 applied to v_1 in its conclusion. In a first phase, we produce a specific fact that contains the information for our desired function application:

$$R(\ldots) \rightarrow F(\ldots, x_3, v_1)$$

In particular, there could be other dependencies that also produce $F(\ldots, x_3, v_1)$ facts. Then, in a second phase, there is one dependency for every function symbol, which applies this function symbol. E.g., for f_3 we have the following dependency:

$$F(\ldots, x_3, y) \rightarrow R(\ldots, f_3(y), \ldots)$$

In this way, we guarantee that for each function symbol, there is only one dependency in which it occurs. ◁

Using these three ideas, it is possible to show that sticky guarded standard Henkin tgds can be used to encode the PCP. Furthermore, through a number of techniques given in detail in the full version, it is possible to further restrict the setting to *linear* Henkin tgds which in total use just two unary function symbols.

Theorem 5.1 *Atomic query answering for sticky linear standard Henkin tgds is undecidable even if only two unary function symbols are allowed in the set of dependencies.*

Another viewpoint of the preceding theorem is that undecidability holds even given just two Henkin tgds, while the rest are full tgds (tgds without existential quantifiers). A slight extension of the above described construction allows us to show undecidability also for nested tgds.

Idea 3$^+$: Nested representation. As we saw in Idea 3, we would like to use Henkin tgds of the form

$$F(\ldots, x_3, y) \rightarrow R(\ldots, f_3(y), \ldots)$$

to apply e.g. function symbol f_3. Unfortunately, this is not the Skolemization of a nested tgd. However, it can be, if we add an additional relation symbol Y as follows:

$$Y(y) \wedge F(\ldots, x_3, y) \rightarrow R(\ldots, f_3(y), \ldots)$$

This is the normalized form of the nested tgd

$$Y(y) \rightarrow \exists z\, [F(\ldots, x_3, y) \rightarrow R(\ldots, z, \ldots)]$$

Hence, we may use nested tgds instead of Henkin tgds (of course, additionally providing tgds that produce corresponding Y facts). Observe that we lose linearity in this way, as we now have two atoms in the antecedent of our normalized nested tgd. However, this is no surprise, as linear nested tgds are just guarded tgds (since linearity prevents nesting) and for guarded tgds we know that query answering is decidable. ◁

Theorem 5.2 *Atomic query answering for sticky guarded simple nested tgds is undecidable even if only two unary function symbols are allowed in dependencies.*

Let us return to the big picture, in particular looking back at the three major "families" of decidable query answering shown in Figure 2. For standard Henkin tgds we have seen that undecidability holds even for sticky and linear, which already rules out decidability for most of the diagram. For nested tgds, we have seen undecidability for sticky and guarded – but not for linear. However, we have discussed in Idea 3$^+$ that query answering is decidable for linear nested tgds.

We have thus drawn the decidability/undecidability in Figure 2 for the two major families of *guarded* and

sticky, but are still missing the well-known *acyclicity* family. Yet, weak acyclicity and more generally finite expansion sets guarantee, as the name suggests, a finite number of finite instances over which queries can be evaluated. Thus it is implicit in [13, 5] that query answering is decidable even for SO tgds. In total, we have now a clear picture of the border between decidability/undecidability in Figure 2.

We have seen in Theorem 5.1 that query answering is undecidable even for linear Henkin tgds. Below we show that decidability can be achieved by imposing a further restriction.

Proposition 5.3 *Atomic query answering is decidable for linear Henkin tgds if the schema is considered as fixed.*

It would be interesting to see if the result can be extended to conjunctive queries. Note that this result is in a sense optimal for atomic queries: Not fixing the schema yields undecidability by Theorem 5.1. Not requiring the dependencies to be linear yields undecidability by the classical result that query answering is undecidable even for tgds [20].

6. Model Checking

In this section, we discuss the model checking problem of our dependency formalisms. Recall that for tgds and SO tgds, the complexity of model checking is already known [17, 14, 25]. We thus pinpoint the data/query/-combined complexity of nested tgds as well as Henkin tgds. Let us first formally define the model checking problem and its variants.

MODEL CHECKING($\mathcal{C}$)
Query: A set $\Sigma \subseteq \mathcal{C}$ of dependencies
Data: A model (database instance) I
Question: Does $I \vDash \tau$ hold?

The data/query/combined complexity of model checking refers to the variant of the problem where the query/data/neither is considered fixed.

We start by considering Henkin tgds and first show NP-completeness in data complexity.

Theorem 6.1 *The model checking problem for Henkin tgds is* NP*-complete in data complexity. Hardness holds even for a single s-t standard Henkin tgd.*

Proof (Sketch) NP-membership is clear, since it holds already for SO tgds [14]. NP-hardness for SO tgds was shown in [14] by a reduction from 3-colorability. However, the resulting SO tgd has the form of the one used in Theorem 4.4, where we showed that it cannot be expressed by a Henkin tgd. We thus show NP-hardness for Henkin tgds by a different reduction from 3-colorability. Let an arbitrary instance of 3-colorability be given by the graph $G = (V_G, E_G)$. We construct an equivalent instance of model checking as (σ, I, J) where σ is a single s-t standard Henkin tgd, I is a source instance and J is a target instance.

Let σ be the following (Skolemized) s-t Henkin tgd

$$V(x) \wedge V(y) \to T(x, y, f(x), g(y))$$

Let $V^I = V_G$ and let T^J be defined as follows

$$E_G \times \{c_1, c_2 \mid c_i \in \{r, g, b\}, c_1 \neq c_2\} \cup$$
$$\{(v, v) \mid v \in V_G\} \times \{c_1, c_2 \mid c_i \in \{r, g, b\}, c_1 = c_2\} \cup$$
$$O \times \{c_1, c_2 \mid c_i \in \{r, g, b\}\}$$

where

$$O = \{(v_1, v_2) \mid v_i \in V_G \wedge v_1 \neq v_2\} \setminus E_G$$

Correctness of the reduction follows from the three lines of the definition of T_J. The first line ensures that both ends of an edge are assigned different colors (where one end's color is given by $f(\cdot)$ and the other's by $g(\cdot)$). The second line ensures that $f(v) = g(v)$ holds for all $v \in V_G$. Finally, the last line ensure that our definition of T_J poses no restriction for any combination not covered by the first two lines (i.e., not an edge and not the same vertex). □

We now show NEXPTIME-completeness of model checking for Henkin tgds in query complexity.

Theorem 6.2 *The model checking problem for Henkin tgds is* NEXPTIME*-complete in query complexity and combined complexity. Hardness holds even for s-t standard Henkin tgds.*

Proof (Sketch) Membership already holds for plain SO tgds [25]. We show hardness by reduction from plain SO tgd model checking, which is known to be NEXPTIME-complete [25]. Let (σ, I, J) be an arbitrary instance of plain SO tgd model checking, where σ is a plain SO tgd, I is a source instance and J is a target instance. In [25], it is implicit that NEXPTIME-hardness holds even if

- $\mathsf{dom}(I) = \mathsf{dom}(J) = \{0, 1\}$
- the arity of predicate symbols is at most 3
- σ consists of a single implication, i.e. is of the form

$$\exists \bar{f}\, \forall \bar{x}\, \varphi(\bar{x}) \to \psi(\bar{f}, \bar{x})$$

Hence, we assume that our given instance satisfies the three conditions given above. Intuitively, σ might violate one of the following conditions of standard Henkin tgds: ($\star$) every occurrence of a function symbol must have the same argument list ($\star$) the arguments of each Skolem term must be pairwise distinct ($\star$) no variable is allowed to occur as argument of two distinct function symbols. We ensure these properties by a sequence of transformations, which first applies some equivalence-preserving simplifications to the plain SO tgd and then chooses an appropriate extension of the database schema to allow for a transformation of the formula into a standard Henkin tgd. □

We thus have identified the precise data, query, and combined complexity of model checking for Henkin tgds. We now proceed to nested tgds. Since nested tgds are a first-order formalism, data complexity is in AC_0 while combined/query complexity is in PSPACE. We show that the latter bound is tight.

Theorem 6.3 *The model checking problem for nested tgds is* PSPACE*-complete in query and combined complexity. Hardness holds even for s-t simple nested tgds.*

Proof (Sketch) Membership is clear, since it already holds for first-order logic. We show hardness by reduction from the well-known PSPACE-complete problem of QBF satisfiability. Let ψ be an arbitrary QBF. W.l.o.g., we may assume

- ψ is in prenex form, with strict quantifier alternation starting with a universal quantifier and ending with an existential quantifier
- the quantifier-free part of ψ is in 3-CNF

We thus may assume that ψ has the form

$$\forall x_1 \, \exists y_1 \, \ldots \, \forall x_n \, \exists y_n \, (c_1 \wedge \ldots \wedge c_n)$$

with $c_i = (l_{i1} \vee l_{i2} \vee l_{i3})$ and l_{ij} is a literal over $\{x_1, \ldots, x_n, y_1, \ldots, y_n\}$. We first define source instance I and target instance J of our model checking problem.

$$I = \{P(1,0), P(0,1)\}$$
$$J = \{Q(1,0), Q(0,1)\} \cup \{C(x_1,x_2,x_3) \mid x \in \{0,1\}\} \setminus \{C(0,0,0)\}$$

The nested tgd τ of the model checking problem is defined as follows:

$$\begin{aligned}
&\forall x_1, \tilde{x}_1 \, P(x_1, \tilde{x}) \rightarrow \exists y_1, \tilde{y}_1 \, Q(y_1, \tilde{y}_1) \wedge \\
&\quad \forall x_2, \tilde{x}_2 \, P(x_2, \tilde{x}) \rightarrow \exists y_2, \tilde{y}_2 \, Q(y_2, \tilde{y}_2) \wedge \\
&\qquad \ldots \\
&\qquad\quad \forall x_n, \tilde{x}_n \, P(x_n, \tilde{x}) \rightarrow \exists y_n, \tilde{y}_n \, Q(y_n, \tilde{y}_n) \wedge \\
&\qquad\qquad \bigwedge_{i \in \{1,\ldots,m\}} C(l^\star_{i1}, l^\star_{i2}, l^\star_{i3})
\end{aligned}$$

where

$$l^\star_{ij} = \begin{cases} x_\alpha, & \text{if } l_{ij} = x_\alpha \\ \tilde{x}_\alpha, & \text{if } l_{ij} = \neg x_\alpha \\ y_\beta, & \text{if } l_{ij} = y_\beta \\ \tilde{y}_\beta, & \text{if } l_{ij} = \neg y_\beta \end{cases}$$

Intuitively, nested tgds miss two features of QBFs: negation and disjunction. Negation is encoded using symbols of the form x_α for positive literals and $\tilde{x}_\alpha$ for negative literals. Through appropriate definition of relation P and Q in I resp. J, the correct behavior of complementary literals is guaranteed. Similarly, using C atoms we encode clauses (i.e., disjunctions) and guarantee correctness by encoding the correct behavior of disjunction in the C relation of J. □

7. Conclusion

In this work, we have investigated the expressive power and computability of using function symbols in tgds. We have seen that function symbols introduced by different forms of dependencies correspond to different structures underlying the domain that one wants to model. For hierarchically structured domains, nested tgds are often the best choice, while Henkin tgds are well-suited for non-hierarchical domains. Nevertheless, sometimes the full power of SO tgds is required. We observed that

- nested tgds share with SO tgds the power of quantifying over more than one tgd part;
- Henkin tgds share with SO tgds the power of allowing non-tree structured Skolem terms.

Yet, our results revealed a deep semantical connection between these two formalisms, namely: every nested tgd can be expressed as a logically equivalent (tree) Henkin tgd. Still, there is a price to pay for simulating nested tgds by Henkin tgds rather than by (arbitrary) SO tgds:

- Converting nested tgds to logically equivalent SO tgds only leads to a linear blow-up.
- Converting nested tgds to logically equivalent (tree) Henkin tgds may lead to a non-elementary blow-up. While not occurring for typical examples as we saw, this may be a problem for particularly complex settings.

Our results showed that each of our dependency classes has particular features that *cannot* be simulated by a "lower" class in our semantical inclusion diagram (Figure 3), since we proved that the diagram is complete and all inclusions are proper. Altogether, we can conclude from our analysis that great care must be taken when modeling a particular domain by tgds so as to balance expressive power, simplicity of the formalism, and size of the resulting set of dependencies. In summary, our analysis of Henkin and nested tgds has provided a deep insight into the versatile features of SO tgds. The results of our study underline the value of SO tgds, which support the distinguished features of both Henkin and nested tgds.

For the model checking problem, we have pinpointed the data/query/combined complexity of nested tgds and Henkin tgds: for Henkin tgds we proved NEXPTIME-completeness of query/combined complexity and NP-completeness of data complexity. Moreover, we showed that nested tgds are PSPACE-complete in query/combined complexity (and are known to be in AC_0 data complexity).

For query answering under various forms of dependencies, we have provided a complete picture of the decidability/undecidability border in Figure 2 for all dependency classes studied here. We have shown that even the slightest deviation from the way that tgds use function symbols in their Skolemized form yields undecidability: this holds even for the weakest extensions of tgds, simple nested tgds or standard Henkin tgds, and even for two of the best-known families for guaranteeing decidability for tgds (guarded and sticky) together.

Hence our analysis has revealed that the islands of decidability are quite small. One such island is that for a fixed schema, at least linear Henkin tgds allow for decidability. Moreover, we have observed that the third major family, weak acyclicity, guarantees decidability of query answering even for SO tgds.

For future work, the most burning question is how to narrow the gaps between the islands of decidability of query answering under Henkin/nested/SO tgds. We will thus analyze known decidable fragments of various logics (such as, e.g., the two-variable fragment) and investigate their applicability to our setting of query answering under tgds. Moreover, we also have to explore new paradigms that are tailor-made for Henkin/nested/SO tgds and that go beyond known decidability criteria for other logics. Beyond that, it would be interesting to see whether frameworks such as dependence logic yield additional suitable subclasses of SO tgds.

Acknowledgements. This work has been supported by the Austrian Science Fund, projects (FWF):P25207-N23 and (FWF):Y698, the Vienna Science and Technology Fund, project ICT12-015, as well as the Engineering and Physical Sciences Research Council (EPSRC), Programme Grant EP/M025268/ "VADA: Value Added Data Systems – Principles and Architecture".

8. References

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

[2] S. Amano, L. Libkin, and F. Murlak. Xml schema mappings. In *PODS*, pages 33–42, 2009.

[3] M. Arenas, P. Barceló, L. Libkin, and F. Murlak. *Relational and XML Data Exchange*. Morgan & Claypool Publishers, 2010.

[4] M. Arenas, J. Pérez, J. L. Reutter, and C. Riveros. The language of plain SO-tgds: Composition, inversion and structural properties. *JCSS*, 79(6):763–784, 2013.

[5] J.-F. Baget, M. Leclère, M.-L. Mugnier, and E. Salvat. Extending decidable cases for rules with existential variables. In *IJCAI*, pages 677–682, 2009.

[6] J.-F. Baget, M. Leclère, M.-L. Mugnier, and E. Salvat. On rules with existential variables: Walking the decidability line. *Artif. Intell.*, 175(9-10):1620–1654, 2011.

[7] P. A. Bernstein and S. Melnik. Model management 2.0: manipulating richer mappings. In *SIGMOD*, pages 1–12, 2007.

[8] A. Blass and Y. Gurevich. Henkin quantifiers and complete problems. *Annals of Pure and Applied Logic*, 32(0):1 – 16, 1986.

[9] A. Calì, G. Gottlob, and M. Kifer. Taming the infinite chase: Query answering under expressive relational constraints. *JAIR*, 48:115–174, 2013.

[10] A. Calì, G. Gottlob, and A. Pieris. Advanced processing for ontological queries. *PVLDB*, 3(1):554–565, 2010.

[11] A. Calì, G. Gottlob, and A. Pieris. Query answering under non-guarded rules in datalog+/-. In *RR*, pages 1–17, 2010.

[12] A. Calì, G. Gottlob, and A. Pieris. Towards more expressive ontology languages: The query answering problem. *Artif. Intell.*, 193:87–128, 2012.

[13] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

[14] R. Fagin, P. G. Kolaitis, L. Popa, and W. C. Tan. Composing schema mappings: Second-order dependencies to the rescue. *ACM TODS*, 30(4):994–1055, 2005.

[15] I. Feinerer, R. Pichler, E. Sallinger, and V. Savenkov. On the undecidability of the equivalence of second-order tuple generating dependencies. *Inf. Syst.*, 48:113–129, 2015.

[16] A. Fuxman, M. A. Hernández, C. T. H. Ho, R. J. Miller, P. Papotti, and L. Popa. Nested mappings: Schema mapping reloaded. In *VLDB*, pages 67–78, 2006.

[17] G. Gottlob and P. Senellart. Schema mapping discovery from data instances. *J. ACM*, 57(2), 2010.

[18] P. Hell and J. Nešetřil. The Core of a Graph. *Discrete Mathematics*, 109:117–126, 1992.

[19] M. A. Hernández, H. Ho, L. Popa, A. Fuxman, R. J. Miller, T. Fukuda, and P. Papotti. Creating nested mappings with Clio. In *ICDE*, pages 1487–1488, 2007.

[20] D. S. Johnson and A. C. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. *JCSS*, 28(1):167–189, 1984.

[21] P. G. Kolaitis, M. Lenzerini, and N. Schweikardt, editors. *Data Exchange, Integration, and Streams*, volume 5 of *Dagstuhl Follow-Ups*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013.

[22] P. G. Kolaitis, R. Pichler, E. Sallinger, and V. Savenkov. Nested dependencies: structure and reasoning. In *PODS*, pages 176–187, 2014.

[23] M. Lenzerini. Data integration: A theoretical perspective. In *PODS*, pages 233–246, 2002.

[24] N. Leone, M. Manna, G. Terracina, and P. Veltri. Efficiently computable datalog programs. In *KR*, 2012.

[25] R. Pichler and S. Skritek. The complexity of evaluating tuple generating dependencies. In *ICDT*, pages 244–255, 2011.

[26] E. L. Post. A variant of a recursively unsolvable problem. *J. Symbolic Logic*, 12(2):255–56, 1946.

[27] E. Sallinger. Reasoning about schema mappings. In *Data Exchange, Integration, and Streams*, pages 97–127, 2013.

[28] B. ten Cate and P. G. Kolaitis. Structural characterizations of schema-mapping languages. In *ICDT*, pages 63–72, 2009.

[29] W. J. Walkoe, Jr. Finite partially-ordered quantification. *J. Symb. Log.*, 35(4):535–555, 1970.