Leveraging Equality Generating Dependencies for Chase Termination

Citation for published version:

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
Abstract. Tuple- and equality-generating dependencies have a wide range of applications in knowledge representation and databases, including ontological reasoning, data exchange, and data cleaning. In such settings, the chase is a central tool for many reasoning tasks. Since the chase evaluation might not terminate and it is undecidable whether it terminates, several termination criteria have been proposed, that is, (decidable) sufficient conditions ensuring termination. One of the main weaknesses of current approaches is the limited analysis they perform on equality-generating dependencies (EGDs).

In this paper, we show that an explicit analysis of EGDs can yield significant benefits and discuss a novel approach along this line.

1 Introduction

Tuple- and equality-generating dependencies (TGDs and EGDs, for short) were introduced for the purpose of database normalization and design, and have seen a revival of interest in the last years, with a wide range of applications in knowledge representation and databases (e.g., ontological reasoning, data exchange, and data cleaning). TGDs (a.k.a. existential rules) and EGDs are the basis of several prominent knowledge representation formalisms, such as Datalog+/- [14], and are closely related to the Horn fragments of the OWL 2 ontology language, as well as to Datalog with function symbols (e.g., see [26, 9, 12, 29, 28, 8, 13, 11, 30]).

In this regard, the chase [5] is a central tool for solving many reasoning tasks. It was originally proposed for classical database problems, such as query optimization, query containment and equivalence, dependency implication, and database schema design [1, 4, 27, 35]. In recent years, it has seen a revival of interest because of a wide range of applications where it plays a central role, such as data exchange, data cleaning and repairing, data integration, and ontological reasoning [19, 6, 3, 2, 15, 16, 21, 20, 25, 31].

The execution of the chase involves inserting tuples possibly with null values to satisfy tuple-generating dependencies, and replacing null values with constants or other null values to satisfy equality-generating dependencies. Specifically, the chase consists of applying a sequence of steps, where each step enforces a dependency that is not satisfied by the current instance. It might well be the case
that multiple dependencies can be enforced and, in this case, the chase picks one nondeterministically. Different choices lead to different sequences, some of which might be terminating, while others might not. This aspect is illustrated in the following example.

Example 1. Consider the set of dependencies \( \Sigma_1 \) below:

\[
\begin{align*}
r_1 &: \ N(x) \rightarrow \exists y \ E(x,y) \\
r_2 &: \ E(x,y) \rightarrow N(y) \\
r_3 &: \ E(x,y) \rightarrow x = y\
\end{align*}
\]

and the database \( D = \{ N(a) \} \). All dependencies are satisfied by \( D \), except for \( r_1 \). Thus, the chase enforces \( r_1 \) by adding \( E(a, \eta_1) \) to \( D \), where \( \eta_1 \) is a (labeled) null value. However, this causes both \( r_2 \) and \( r_3 \) to be violated. Suppose the chase chooses to enforce \( r_2 \) by adding \( N(\eta_1) \). Now \( r_1 \) is not satisfied again, while \( r_3 \) continues to be violated. Suppose the chase chooses to enforce \( r_1 \). Then, similar to the first step, \( E(\eta_1, \eta_2) \) is added to the current instance, and this causes \( r_2 \) to become violated again. It is easy to see that repeatedly enforcing first \( r_1 \) and then \( r_2 \) yields an infinite chase sequence that introduces an infinite number of facts: \( N(\eta_2), E(\eta_2, \eta_3), N(\eta_3), \ldots \).

However, by enforcing first \( r_1 \) and then \( r_3 \), we get a terminating chase sequence. Specifically, enforcing \( r_1 \) adds \( E(a, \eta_1) \) to \( D \). Then, the application of \( r_3 \) updates the null value \( \eta_1 \) to \( a \). At this point, no further dependency needs to be enforced, and the chase terminates with the resulting database being \( \{ N(a), E(a, a) \} \).

The importance of the chase in many applications is due to the fact that several problems (e.g., checking query containment under dependencies, checking implication of dependencies, computing solutions in data exchange, and computing certain answers in data integration) can be solved by exhibiting a universal model, and the chase computes a universal model, when it terminates [17]. Roughly speaking, a model for a database and a set of dependencies is a finite instance that includes the database and satisfies the dependencies. A universal model is a model that can be “mapped” to every other model—in a sense, it represents the entire space of possible models. Universal models are slight generalizations of universal solutions in data exchange [19], and can be used to compute them. Moreover, the certain answers to a conjunctive query in the presence of dependencies can be computed by evaluating the query over a universal model (rather than considering all models). Answering conjunctive queries over a set of facts extended with existential rules is a prominent problem in ontological reasoning. Other applications of universal models (e.g., dependency implication and query containment under dependencies) can be found in [17].

Thus, finding a universal model is a central problem in many applications and, once again, the chase is a tool to solve it, provided that it terminates. As a consequence, checking whether the chase terminates becomes a central problem, but unfortunately, it is an undecidable one [22]. To cope with this issue, several “termination criteria” have been proposed, that is, (decidable) sufficient conditions ensuring chase termination.

Indeed, as illustrated in Example 1 above, when we talk about chase termination, it is important to distinguish between two problems: checking whether
all chase sequences are terminating, and checking whether there is at least one terminating chase sequence. Most of the work in the literature has focused on the problem of checking if all chase sequences are terminating, independently of the considered database. However, since in many applications the ultimate goal is to compute a universal model, checking for the existence of a terminating chase sequence and constructing it suffices for the purpose.

Furthermore, the weaker requirement of checking for the existence of a terminating chase sequence can be profitably leveraged to identify more sets of dependencies for which we can compute a universal model. For instance, the set of dependencies $\Sigma_1$ of Example 1 might be identified by a criterion ensuring termination of at least one chase sequence. However, every criterion requiring all chase sequences to be terminating will not recognize $\Sigma_1$, thereby providing no information about whether we can compute a universal model.

Despite the significant body of work in this area, there are still large classes of dependencies for which the chase is not applicable as termination cannot be statically established. One weakness of current approaches is that the analysis of EGDs is limited or absent. In fact, more general approaches, such as super-weak acyclicity [36], semi-dynamic approaches MFA and MSA [24], and rewriting approaches [32–34], were meant to guarantee termination of TGDs only. Other approaches, such as weak acyclicity [19] and safety [37], guarantee the termination of a set of TGDs and EGDs, but do not analyze EGDs, which leads them to impose strong conditions on TGDs. Firing relations among dependences used in stratification-based approaches [17, 37, 33, 34] consider EGDs in a limited way. To mitigate the aforementioned issues, an “indirect” way of dealing with EGDs was proposed in [23, 36], where a set $\Sigma$ of TGDs and EGDs is rewritten into a set $\Sigma'$ containing only TGDs, and termination analysis is carried out on $\Sigma'$. The aim is to “simulate” the behavior of the EGDs by means of TGDs. While these preprocessing steps ensure soundness, i.e., if all chase sequences of $\Sigma'$ are terminating then all chase sequences of $\Sigma$ are terminating, they are not complete, i.e., the implication in the opposite direction does not hold.

Treating EGDs as first-class citizens is very important, as they are among the most popular classes of dependencies in real applications, playing a critical role in maintaining data integrity, query optimization and indexing, and schema design [18]. For instance, functional dependencies can be expressed by EGDs. In very simple scenarios, such as Example 1 above, current termination criteria are not able to say whether a universal model can be found. As a further scenario, Example 2 shows a simple set of dependencies for which all chase sequences are terminating, but there is no terminating chase sequence for the set of dependencies obtained from the EGD simulation.

In this paper, we show that an explicit analysis of EGDs can yield significant benefits, and discuss a novel approach able to perform this kind of analysis.

2 Dealing with EGDs

In this section, we discuss several issues that arise in the presence of EGDs.
Example 2. Consider the following set of dependencies

\[
\Sigma \ni \begin{align*}
\text{TGDs} & : A(x) \land B(x) \rightarrow C(x) \\
\text{EGDs} & : B(x) \land B(y) \rightarrow x = y
\end{align*}
\]

The substitution-free simulation works as follows:

1. The TGDs below (equality-axioms) are added to \( \Sigma \):

\[
\begin{align*}
a_1 : Eq(x, y) & \rightarrow Eq(y, x) \\
a_2 : Eq(x, y) \land Eq(y, z) & \rightarrow Eq(x, z) \\
a_3,1 : A(x) & \rightarrow Eq(x, x) \\
a_3,2 : B(x) & \rightarrow Eq(x, x) \\
a_3,3 : C(x) & \rightarrow Eq(x, x)
\end{align*}
\]

2. Every occurrence of \( x = y \) in \( \Sigma \) is replaced with \( Eq(x, y) \). In our case, this affects \( r_4 \) and \( r_5 \) only, which are replaced with:

\[
\begin{align*}
r'_4 & : A(x) \land A(y) \rightarrow Eq(x, y) \\
r'_5 & : B(x) \land B(y) \rightarrow Eq(x, y)
\end{align*}
\]
3. Dependency $r_1$, which contains multiple occurrences of $x$ in the body, is (non-deterministically) replaced with one of the following two dependencies, where one of the two occurrences of $x$ is replaced with $x_2$, and the atom $Eq(x, x_2)$ is added to the body:

$$r_1': A(x_2) \land B(x) \land Eq(x_2, x_2) \rightarrow C(x)$$
$$r_2': A(x) \land B(x_2) \land Eq(x, x_2) \rightarrow C(x)$$

Notice that the only dependencies that remain unchanged are $r_2$ and $r_3$ and that no EGDs occur in the resulting set of dependencies.

Although somehow left implicit in [23, 36], the natural simulation and the substitution-free simulation ensure the desirable soundness property: if $\Sigma' \in CT_\forall^*$, then $\Sigma \in CT_\forall^*$, for $c \in \{\text{obl, sobl, std}\}$. The natural question now is whether these simulations are also complete, that is, if the implication in the opposite direction holds. The answer is negative for both approaches.

For example, the set of dependencies $\Sigma_3$ of Example 2 above belongs to $CT_\forall^*$ (and thus belongs to $CT_\forall^*$ too), but any of its substitution-free simulations is not even in $CT_\forall^*$, for every $c \in \{\text{obl, sobl, std}\}$. The problem is that the simulation of EGDs by means of TGDs is not able to fully capture the specific behavior of EGDs, which replace null values (with constants and other null values). This aspect is not faithfully modeled by storing the information that a null value is equal to a constant or to another null value.

Dealing with EGDs needs some care. In some cases the presence of EGDs allows us to have a terminating $c$-chase sequence when the set consisting only of the TGDs does not have one; at the same time, the opposite case can occur, that is, in the presence of EGDs there is no terminating $c$-chase sequence while the set consisting only of the TGDs does have one, where $c$ can be one of $\{\text{obl, sobl, std}\}$. The following two examples show such cases.

**Example 3.** Consider the set of dependencies $\Sigma_1$ of Example 1 and the database $D = \{N(a)\}$. There is no terminating $c$-chase sequence of $D_1$ with the set of TGDs $\Sigma_1' = \{r_1, r_2\}$, for every $c \in \{\text{obl, sobl, std}\}$. In fact, it is easy to see that an infinite number of facts is introduced: $E(a, \eta_1), N(\eta_1), E(\eta_1, \eta_2), \ldots$. However, the addition of the EGD $r_3$ allows us to have a terminating $c$-chase sequence, obtained by enforcing first $r_1$ and then $r_3$, and whose result is the universal model $\{N(a), E(a, a)\}$.

**Example 4.** Consider the set of dependencies $\Sigma_4$ below:

$$r_1: N(x) \rightarrow \exists y \exists z \ E(x, y, z)$$
$$r_2: E(x, y, y) \rightarrow N(y)$$
$$r_3: E(x, y, z) \rightarrow y = z$$

For every database $D$, every $c$-chase sequence of $D$ with the set of TGDs $\Sigma_4' = \{r_1, r_2\}$ is terminating, for every $c \in \{\text{obl, sobl, std}\}$. On the other hand, there is no terminating $c$-chase sequence of $D = \{N(a)\}$ with $\Sigma_4$, as an infinite number of facts is introduced: $E(a, \eta_1), N(\eta_1), E(\eta_1, \eta_2, \eta_2), N(\eta_2), \ldots$. 


3 A Rewriting Approach

In this section, we illustrate a decidable sufficient condition for a set of dependencies to be in $\text{CT}^{\text{std}}_\exists$. It is an algorithm which takes as input a set of dependencies, and gives as output a set of adorned dependencies and a boolean value. The aim of the algorithm is twofold: (i) it defines a termination criterion on its own—on the basis of the boolean value returned by the algorithm; and (ii) it can be combined with other termination criteria to enhance them, in that (strictly) more sets of dependencies in $\text{CT}^{\text{std}}_\exists$ can be identified by using our algorithm in conjunction with a termination criterion—this is achieved by analyzing the set of adorned dependencies returned by the algorithm.

Thus, the input is a set of dependencies $\Sigma$, while the output is a set of adorned dependencies $\Sigma^\mu$ along with a boolean value $\text{Acyc}$. More specifically, if $\text{Acyc}$ is false, then a form of cyclicity has been detected; otherwise, for every database $D$, there is a terminating standard chase sequence of $D$ with $\Sigma$. As for the second aim of the algorithm, the adorned set of dependencies $\Sigma^\mu$ given as output can be used as follows: a sufficient condition for checking membership in $\text{CT}^{\text{std}}_\exists$ is applied to $\Sigma^\mu$ rather than $\Sigma$. If $\Sigma^\mu$ satisfies the condition, then the original set of dependencies $\Sigma$ is in $\text{CT}^{\text{std}}_\exists$.

The basic idea of the algorithm is to produce adorned dependencies from the original ones by keeping track of what facts can be derived by a chase execution and how terms are derived. When adornning dependencies, the algorithm’s strategy is to adorn first full dependencies, and to adorn existentially quantified dependencies only when no further full dependency can be adorned. This is iterated as long as new adorned dependencies can be derived. EGDs are leveraged to see if free symbols can be changed.

The algorithm makes use of adornment definitions, which are expressions of the form $f_i = f^r_z(\alpha)$, whose intuitive meaning is that $z$ is an existentially quantified variable in the head of the TGD $r$ and $\alpha$ is an adornment for the body of $r$. Roughly speaking, adornment definitions are some sort of provenance information used to keep track of the “history” of a null.

The following example gives the basic idea of how the technique works.

Example 5. Consider the set of dependencies $\Sigma_1$ of Example 1. Initially, the following two adorned dependencies, mapping unadorned atoms to atoms adorned with strings of $b$’s, are added to $\Sigma_1^\mu$:

$$
\begin{align*}
s_1 &: N(x) \rightarrow N^b(x) \\
s_2 &: E(x, y) \rightarrow E^{bb}(x, y)
\end{align*}
$$

The algorithm then proceeds by adorning full dependencies and adds the following adorned dependencies to $\Sigma_1^\mu$:

$$
\begin{align*}
s_3 &: E^{bb}(x, y) \rightarrow x = y \\
s_4 &: E^{bb}(x, y) \rightarrow N^b(x)
\end{align*}
$$

Next, the existentially quantified dependency (namely, $r_1$) is adorned and the following adorned dependency is added to $\Sigma_1^\mu$:

$$
\begin{align*}
s_5 &: N^b(x) \rightarrow \exists y \ E^{b/1}(x, y)
\end{align*}
$$
Moreover, the following adornment definition is derived $f_1 = f_y^1(b)$. After that, the algorithm starts considering full dependencies again. By adorning the EGD $r_3$, the following adorned dependency is obtained, which is added to $\Sigma_1^\mu$:

$$s_6 : E^{bf_1}(x, y) \rightarrow x = y$$

The adorned EGD $s_6$ is analyzed by the algorithm, which replaces $f_1$ with $b$ in $\Sigma_1^\mu$. The resulting set of adorned dependencies is $\Sigma_1^\mu = \{s_1, s_2, s_3, s_4, s_5, s_6\}$, where $s'_5$ is derived from $s_5$ by replacing $f_1$ with $b$, that is, $s'_5 : N^b(x) \rightarrow \exists y E^{bb}(x, y)$.

At this point, no further dependencies can be adorned and the algorithm terminates by returning the value $Acyc = true$ along with $\Sigma_1^\mu$.

References
