Rewriting the Infinite Chase (with supplementary material as an appendix)

Michael Benedikt
Oxford University
Oxford, United Kingdom
michael.benedikt@cs.ox.ac.uk

Maxime Buron
Oxford University
Oxford, United Kingdom
maxime.buron@cs.ox.ac.uk

Stefano Germano
Oxford University
Oxford, United Kingdom
stefano.germano@cs.ox.ac.uk

Kevin Kappelmann
Technical University of Munich
Munich, Germany
kevin.kappelmann@tum.de

Boris Motik
Oxford University
Oxford, United Kingdom
boris.motik@cs.ox.ac.uk

ABSTRACT

Guarded tuple-generating dependencies (GTGDs) form a natural extension of description logics and referential constraints. Although reasoning problems with GTGDs have been known to be decidable for several decades, there has been little work on concrete algorithms and even less on implementation. We address this gap by revisiting the approach to atomic query answering over GTGDs via backwards reasoning or Datalog-rewriting. We show how sound rewriting-based approaches can be motivated from the "forward reasoning approach" — that is, via the chase. We provide a set of algorithms for rewriting that represent different ways of simulating the chase, along with optimizations of these approaches. We then present an experimental analysis of our methods, based on a combination of synthetic and manually-curated benchmarks.

PVLDB Reference Format:
Michael Benedikt, Maxime Buron, Stefano Germano, Kevin Kappelmann, and Boris Motik. Rewriting the Infinite Chase (with supplementary material as an appendix). PVLDB, 15(1): XXX-XXX, 2022.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/KRR-Oxford/Guarded-saturation.

1 INTRODUCTION

Reasoning with dependencies has played a major role in database theory for decades. Dependencies can be used to describe semantic restrictions on sources, mapping rules between data sources and virtual data objects in data integration, and semantic rules on virtual data that allow new fact to be derived. A fundamental computation problem associated with dependencies is *query answering*: given a query Q, as an existentially quantified conjunction of atoms (a *conjunctive query*), a collection of facts I, and a set of dependencies Σ, find all the answers to Q that can be derived from I via reasoning with Σ. The problem arises in data integration, where the dependencies describe relationships between heterogeneous sources and a global schema representing virtual tables for the integrated data, and may also describe constraints on the integrated data. Indeed, query answering with dependencies has long been seen as a key component in a declarative data integration system [21, 28]. It also arises in querying using views or access patterns [18, 22, 31].

For general classes of dependencies, such as tuple-generating dependencies (TGDs), query answering is undecidable. Therefore, much early work focused on dependencies for which query answering is decidable. One line of work has focused on dependencies with *terminating chase*: those for which forward-reasoning by inferring all new facts from I will terminate in a finite number of steps. Weakly-acyclic dependencies are perhaps the best-known class with terminating chase. A different strand focuses on dependencies where *backwards reasoning terminates*; these are often known as *rewritable* classes. An attractive class of dependencies that emerged in the last decade are the guarded tuple-generating dependencies (GTGDs). Guarded TGDs allow one to express simple referential constraints on source instances, common mapping rules between sources and targets, and target constraints in common description logic and ontology languages. Query-answering for GTGDs has long been known to be decidable [29]. However, the problem is challenging in that GTGDs do not in general have terminating chase and they are not first-order rewritable.

Example 1.1. Consider a schema for a virtual knowledge base. There are relations capturing basic person/place/thing concepts (e.g. $\mathsf{Person}(x)$) and their relationships (e.g. $\mathsf{ChildOf}(x, y)$), and also relations representing aggregated information from source database about connections, contacts, and a contact directory. The schema has constraints that capture the semantics of these relations. There could be axioms about general concepts such as people:

$$\mathsf{Person}(x) \rightarrow \exists y\ \mathsf{Person}(y) \land \mathsf{ChildOf}(x, y)$$

And there could be axioms stating that for each contact, any referring entity is stored in a directory, and that entries in the directory are also contacts:

$$\mathsf{Connection}(\mathsf{pid}, \mathsf{pname}, \mathsf{refpid}, \mathsf{refpname}, \mathsf{date}) \land \mathsf{Contact}(\mathsf{pid}, \mathsf{pname})$$
$$\rightarrow \exists \mathsf{entrydate}\ \mathsf{Directory}(\mathsf{refpid}, \mathsf{refpname}, \mathsf{entrydate})$$
$$\mathsf{Directory}(\mathsf{entryid}, \mathsf{entryname}, \mathsf{entrydate}) \rightarrow \mathsf{Contact}(\mathsf{entryid}, \mathsf{entryname})$$

Proceedings of the VLDB Endowment, Vol. 15, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

We give here only a fragment of what such a constraint set would look like. There would also be "mapping rules" relating data in local sources to these target constraints.

Consider a query that asks for the list of contacts. Intuitively, we should apply first the source-to-target rules to generate the integrated database, and then recursively apply the target constraints. But it is not clear when we might terminate the latter process: the first set of rules naïvely generates an infinite chain of ChildOf facts. ◁

There are several ways to prove that GTGD query-answering is decidable. One can argue that it is enough to look at answers that hold on all *tree-like models* – those structures that can be coded by trees. Using this, one reduces to the problem of emptiness of a suitable tree automata. The approach is implicit in the original proof of decidability for query answering in [29]. A variant is to create a Tableau calculus for the logic, which tries to build a tree-like model, using a form of *blocking* to ensure termination. This approach is widely-applied in the setting of description logics [6], and was later lifted to the setting of guarded logics [23]. The first method has two drawbacks: building the automaton is extremely expensive, and the non-emptiness test is both complex and expensive. The second approach is limited by the complexity of implementing blocking, which is difficult even in the simpler setting of description logics.

An alternative approach is based on *Datalog-rewriting*. Starting from Σ alone, we create a set P_Σ of TGDs with no existentials in the head. For the purposes of query answering, these can be identified with rules in the common database query language *Datalog*. In this work, we will focus on the case of atomic ground queries Q, where a rewriting is query-independent, and thus denote this as a *Datalog-rewriting of the dependencies.* Formally, P_Σ should have the property that for any dataset I, P_Σ produces the same ground consequences as Σ does. Thus, if Q is atomic, it suffices to generate the ground consequences of P_Σ and then evaluate Q on the result. Since P_Σ consists of full TGDs, finding the ground consequences can be done by a simple algorithm, PTIME in the data. Indeed, we can find these consequences via any standard Datalog engine. This approach to Guarded TGD query answering has its origins in work of Marnette [30], but has been extended to wider classes of constraints [8, 20] and refined to deduce stronger properties of the rewriting [9]. In principle, the Datalog-rewriting approach has the advantage of being scalable in the size of the input instance. While the approach has been implemented for description logics [24, 25], concrete algorithms in the setting of GTGDs have not yet appeared.

Example 1.2. Returning to Example 1.1, intuitively, we can replace the *GTGDs* with the following derived Datalog rule:

$$\mathsf{Connection}(\mathsf{pid}, \mathsf{pname}, \mathsf{refpid}, \mathsf{refpname}, \mathsf{date}) \wedge$$
$$\mathsf{Contact}(\mathsf{pid}, \mathsf{pname}) \rightarrow \mathsf{Contact}(\mathsf{refpid}, \mathsf{refpname})$$

This rule can be combined with the source-to-target mapping rules in a standard way to get a Datalog program computing the IDs and names of contacts directly from the source databases. Note that the rule involving ChildOf and Person does *not* end up contributing to the rewriting at all in this case; however, detecting which rules contribute is highly non-trivial. ◁

A number of issues quickly arise when considering Datalog-rewritings.

How can we see what Datalog rules we need to create to get a complete rewriting? Although the prior literature includes several algorithms for rewriting, it is hard to see how they were arrived or why they should work. Previous approaches to proving correctness of Datalog rewriting for description logics [32] relied on general completeness arguments for resolution, which are difficult to customize to the setting of GTGDs, or rely on complex arguments for more general guarded logics (e.g. the "resolution game" of [17]). Our first contribution is to *relate Datalog-rewriting approaches to the standard technique for analyzing "data-exchange" or "forward-reasoning" approaches, via the chase.* For this we will need a new analysis of the chase for Guarded TGDs, which will allow us to state a correctness criterion for rewriting based on a specialized chase. Our chase-based approach makes rewriting algorithms easier to understand and prove correct, and allows us to discover new rewriting algorithms.

What does the space of rewriting algorithms look like? When creating the desired full TGDs P_Σ, inductively we need to create a larger set of consequences of Σ. We will show that there are actually a number of alternatives for these consequence sets. One approach only adds new full TGDs alone, each of which directly contributes to the final rewriting. Alternative approaches require adding new TGDs that have existential quantifiers in the head ("non-full TGDs"). Still others generate TGDs with no existentials in the head, but with special function symbols – "Skolem functions" in the body and head. We introduce several algorithms, showing in each case how they related to the chased-based framework mentioned previously. We provide theoretical worst-case guarantees on the performance of each algorithm.

How do we make these rewriting algorithms scalable? Previous work on implementation of Datalog-rewriting has taken place only in the more restricted setting of description logics [25, 32], and in the more general setting of first order theorem proving [39]. We build on this prior line to deal with the GTGD rewritings introduced in the paper. Our toolbox will include indexing of rules, pruning of the search space, and rule transformation. *To the best of our knowledge, we provide the first look at optimization and implementation for GTGD rewriting.*

How do we evaluate Datalog-rewriting algorithms? We provide a benchmark for comparing GTGD query-answering algorithms, and we evaluate our methods on this benchmark. To the best of our knowledge this is the first work on evaluating query answering for Guarded TGDs in general, and on evaluating Datalog rewriting on any class of TGDs.

Summary of contributions. In this work we give a general account of Datalog-rewriting in the presence of GTGDs. We contribute a theoretical framework for understanding, motivating, and showing completeness of rewritings. We provide new algorithms rewriting, establishing bounds on their performance and how they compare to each other. We complement this theoretical analysis with an investigation of how implementation techniques from theorem proving extend to the setting of Guarded TGDs. Finally, we introduce a benchmark for rewriting techniques and evaluate our methods on it. *For space reasons, many details of proofs and algorithms are deferred to the supplementary materials, while our implementation and a fuller view of experiments can be found at [11].*

2 RELATED WORK

First-order rewritings were developed in description logics, leading to the DL-Lite family of ontology languages [16]. The concept was also introduced into higher-arity languages for database mapping rules and target constraints [15]. In the context of description logics, [25] consider computing Datalog-rewritings, which are often called *saturations*. The KAON2 system [32] is a canonical example of the saturation/Datalog-rewriting approach within description logics. Approaches closely related to Datalog-rewriting have been considered for the Guarded Fragment, a logic much richer than GTGDs. [19]. Datalog rewriting in the context of GTGDs was first considered in [30]. Later algorithms for rewriting [9, 20] allow for richer sets of constraints (e.g. the frontier-guarded TGDs of [8]) while showing that the rewriting of a set of GTGDs can be taken to be a set of full GTGDs – roughly speaking, the rewriting is in Guarded Datalog rather than merely in Datalog, and similarly for frontier-guarded TGDs. But the focus of these works was complexity bounds, not to provide a useful algorithm. To the best of our knowledge, there has not been any implementation of query answering for GTGDs, even for ground facts. There have been several implementations of query answering for classes of dependencies where the chase terminates: an overview can be found in [12]. There have been prototypes supporting classes where the chase does not terminate. [42] supported linear TGDs, [10] supports the *Warded fragment*, and [3, 43] support Shy, weakly acyclic and separable TGDs; all of these classes are incomparable with GTGDs.

The chased-based approach to rewriting presented here derives from a specialized chase used for analysing rewriting with access patterns in [4], and from the later unpublished masters thesis [27]. [27] also introduces a variant of the one-pass chase presented here, and generalizes it to the broader context of disjunctive TGDs.

3 PRELIMINARIES

We consider *schemas* consisting of a set of relations, each associated with an arity (a non-negative number). We will assume a fixed infinite set of *base values* Vals. A *base instance* of a schema is an assignment of each relation in the schema of arity n to a collection of n-tuples from Vals, while an *instance* assigns arbitrary tuples (the *values* of the instance). Given an instance I and relation R, we let $\mathsf{I}(R)$ be the n-tuples assigned to R in I. A *fact* consists of a relation R of arity n and an n-tuple $\vec{c}$. We write such a fact as $R(\vec{c})$, and for each i say that it *contains* c_i. An instance can equivalently be thought of as a set of facts.

Let vars be an infinite set, the *variables*. A *relational atom* (over vars and Vals) consists of a relation R of arity n and an n-tuple $\vec{t}$, where each t_i is a *term* built up from variables and base values. An atom will be written $R(\vec{t})$. We write $\mathsf{consts}(R)$ for the set of base values in R.

A conjunctive query (CQ) is a formula of the form $\exists \vec{y} \ \bigwedge_i A_i$ where each A_i is an atom. A CQ with no free variables is a Boolean CQ (*BCQ*). Note that a fact is a special case of a BCQ.

For any logical formula γ with free variables $x_1, \ldots, x_n$ and an instance I, a *substitution* for γ is a mapping taking each x_i to a term, while a *ground substitution* maps each variable to a ground term. Given any logical formula γ and substitution σ, we can apply σ to γ to get a new formula $\sigma(\gamma)$ where each x_i is replaced by $\sigma(x_i)$.

Given a conjunction of atoms $\bigwedge_i A_i$, a ground substitution σ is a *homomorphism* of the conjunction into I if, for each A_i, the ground atom $\sigma(A_i)$ holds in I (that is, it is a fact of I). A homomorphism of a CQ Q into I is a homomorphism of the conjunction of atoms in Q into I. We also write $\mathsf{I}, \sigma \models Q$ if σ is a homomorphism of Q, and for Q Boolean we write $\mathsf{I} \models Q$ if there is a σ with $I, \sigma \models Q$.

Tuple-generating Dependencies. A *tuple generating dependency* (TGD) is a first-order formula of the form: $\forall \vec{x}[\beta \rightarrow \exists \vec{y} \ \eta]$ where β and η are conjunctions of atoms, β has free variables $\vec{x}$, and η uses free variables that come from $\vec{x} \cup \vec{y}$. β is referred to as the *body* and $\exists \vec{y} \ \eta$ as the *head*. The variables that occur on both sides, a subset of $\vec{x}$, are said to be *exported* in the TGD. TGDs whose head contains no existential quantifiers are denoted as *full rules* or *Datalog rules*; in this paper these terms will be used interchangeably. We will assume that our TGDs τ are in *head normal form*, that is each τ is in Datalog or every head atom contains at least one existential variable. This normal form is easy to achieve, and for any of our routines that do not preserve this normal form, we implicitly normalize as a post-processing step. The *headwidth* (hwidth) of a TGD is the number of variables in the head, while the *bodywidth* (bwidth) is the number in the body. The *width* (width) is the maximum of hwidth and bwidth. We extend these definitions to a set of TGDs by taking the maximum over all *TGDs* in the set. We sometimes drop universal quantifiers in TGDs. A *Guarded TGD* (GTGD) is a TGD where a single atom of β contains all the variables of β.

The notion of a rule being satisfied in an instance is the usual one in first-order logic. It can be restated as saying that every homomorphism of β extends to a homomorphism of η. A ground fact F is *entailed* by a collection of facts I and a set of rules Σ if for every instance I' containing I and satisfying Σ, I' contains F. We write $\mathsf{I}, \Sigma \models F$ when this happens. Given a set of rules Σ and a base instance I, the *ground closure of* I *under* Σ is the set of facts using only values from I that are entailed by I and Σ. The *query answering problem* is to decide whether a given CQ Q is entailed by a base instance I and Σ. A special case – the *fact entailment problem* – is to decide whether a given fact F using only values in I is entailed by a base instance I and rules Σ. That is, whether F is in the ground closure of I under Σ.

Unification. An important step in some of our algorithms will be the unification of atoms that may contain terms with function symbols [38].

Given two substitutions ρ, ρ', we write $\rho\rho'$ for the composition of the substitutions; that is, $\rho\rho' := \rho' \circ \rho$. A *unifier* of atoms $A_1, \ldots, A_n$ and $B_1, \ldots, B_n$ is a substitution θ satisfying $\theta(A_i) = \theta(B_i)$ for $1 \leq i \leq n$. A unifier θ of atoms $A_1, \ldots, A_n$ and $B_1, \ldots, B_n$ is called a *most general unifier* (mgu) if for any unifier ρ of $A_1, \ldots, A_n$ and $B_1, \ldots, B_n$, there is a substitution δ such that $\rho = \theta\delta$.

It is well known that mgus are unique and that the mgu of atoms $A_1, \ldots, A_n$ and $B_1, \ldots, B_n$ can be obtained in time $O\big(\sum_{i=1}^{n} |A_i| + |B_i|\big)$, where $|A|$ is the encoding size of an atom A: see, for example, [35]. Unification is a key component in *resolution* of two logical formulas to get a new formula that is a consequence of the first two. In our case we will resolve rules $\tau = \beta \rightarrow \eta$ and $\tau' = \beta' \rightarrow \eta'$ to get a new rule, where some atoms in η are unified with atoms in β'. We will not need the general definition of resolution from [38], since we will present specialized resolution steps for our setting.

3.1 A primer on query answering with GTGDs

The previous proofs of decidability for query answering over Guarded TGDs were based on the idea of "tree-like chase sequences". It combines ideas from more general guarded logics [5, 41] with work on reasoning with referential constraints in database [26], and we review the core idea here.

To witness an entailment $A, \Sigma \models F$, we can use a *tree-like chase sequence*, i.e. a sequence $T_0, \ldots, T_n$ of *chase trees*. A chase tree T_i in such a sequence consists of a tree structure with a function $\mathsf{FactsOf}_i$ that maps each node of $v \in T_i$ to a collection of facts $\mathsf{FactsOf}_i(v)$.

In a tree-like chase sequence $T_0, \ldots, T_n$, consecutive chase trees will be linked by two kinds of steps. First, there will be *chase steps*, which add facts to the tree, possibly in a new node. A *trigger* for a TGD $\tau = \beta \rightarrow \exists \vec{y}\, \eta \in \Sigma$ in tree T_i is a homomorphism σ of β into $\mathsf{FactsOf}_i(v)$ for some node v. Applying a *(tree-like) chase step* for τ and σ in T_i creates T_{i+1}, extending T_i with new facts. If τ is a Datalog rule, then we simply add the facts in the head to v. If τ is non-full, T_{i+1} is T_i augmented with a new child c of v, with $\mathsf{FactsOf}_{i+1}(c)$ including $\sigma'(\eta(\vec{x}, \vec{y}))$, where σ' is an extension of σ and $\sigma'(y_i)$ is a fresh value that does not occur in T_i for each $y_i \in \vec{y}$. We also include in the child any fact of $\mathsf{FactsOf}_i(v)$ which is guarded by a fact in $\sigma'(\eta(\vec{x}, \vec{y}))$. A fact G *guards* another fact F if $\mathsf{consts}(F) \subseteq \mathsf{consts}(G) \cup \mathsf{consts}(\Sigma)$. In the above case we say that *τ based on σ fires in T_i*, or *τ fires in T_i* if there is some trigger for which τ fires in T_i. Thus, the chase steps add new facts that make τ hold for this substitution and put them either in v or in a child node of v. A step from T_i to T_{i+1} can also be a *propagation step*, which does not change the underlying tree but just copies facts from one node to another, i.e. it modifies $\mathsf{FactsOf}$ while maintaining the other components. We are only allowed to propagate a fact $R(\vec{c})$ to some other node v' of the tree such that v' has a guard for $R(\vec{c})$ in $\mathsf{FactsOf}_i(v')$.

A sequence as above is a *tree-like chase proof* of $A, \Sigma \models Q$ if the initial tree structure T_0 consists of a single root node r, with $\mathsf{FactsOf}(r) = A$, and the final tree structure satisfies Q.

Prior work on query answering reduces it to finding tree-like chase proofs:

Proposition 3.1 ([29]). *For GTGDs Σ, facts A, and a BCQ Q, $A, \Sigma \models Q$ if and only if there is a tree-like chase proof that witnesses this.*

Example 3.2. Given instance $I = \{R(c, d)\}$ and a set of GTGDs Σ

$$\begin{array}{ll}
R(x_1, x_2) \rightarrow \exists y\, S(x_1, y), & R(x_1, x_2) \rightarrow \exists y\, T(x_1, x_2, y), \\
T(x_1, x_2, x_3) \rightarrow \exists y\, U(x_1, x_2, y), & U(x_1, x_2, x_3) \rightarrow P(x_2), \\
T(x_1, x_2, x_3) \wedge P(x_2) \rightarrow M(x_1), & S(x_1, x_2) \wedge M(x_1) \rightarrow \exists y\, N(x_1, y),
\end{array}$$

the sequence

$$\begin{array}{lll}
I_0 = \{R(c, d)\}, & I_1 = I_0 \cup \{S(c, d_1)\}, & I_2 = I_1 \cup \{T(c, d, d_2)\}, \\
I_3 = I_2 \cup \{U(c, d, d_3)\}, & I_4 = I_3 \cup \{P(d)\}, & I_5 = I_4 \cup \{M(c)\}, \\
I_6 = I_5 \cup \{N(c, d_4)\} & &
\end{array}$$

is a chase sequence for I and Σ. A corresponding tree-like chase is depicted in Figure 1. ◁

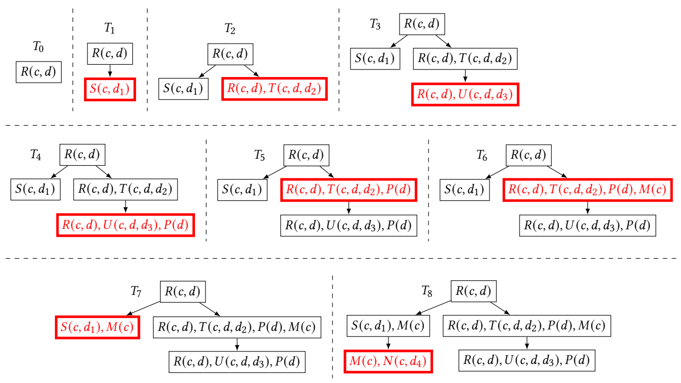


Figure 1: Tree-like chase for Example 3.2; the node that has just been modified is shown in red.

There are several ways to search for tree-like chase proofs. [29] gives a bound on the height of a tree within a tree-like proof that can witness that a fact is entailed. In the case of atomic query answering, it suffices to bound the number of possible contents of a node that can appear on a given path, up to isomorphism over the base structure. One can then generate trees up that bound, cutting off the search when the bound is exceeded. One can also build a tree automaton $\mathcal{A}_{\Sigma,A}$ that accepts all chase trees that can appear in such a sequence for a given set of GTGDs Σ and initial instance A. The labels of these trees will code the set of facts at a given node, along with the overlap relations between neighboring nodes. The automaton-based approach also requires building an automaton $\mathcal{A}_Q$ that checks whether Q holds in a chase tree. With these two automata in place, one can use standard automata-theoretic methods to see if $\mathcal{A}_Q$ and $\mathcal{A}_{\Sigma,A}$ both accept the same tree. Notice that as A grows large the potential number of chase trees rooted at A, even restricting to a given height, is quite large. Thus searching for tree-like chase proofs is not practical. Likewise the automata used will be extremely large — indeed, even the number of labels that the automata will use will be huge. Still, the tree-like chase will motivate our rewriting approaches.

Rewriting. An alternative approach to chasing is *Datalog rewriting*. In the first step we will replace Σ with a set of Datalog rules which have the same impact on the ground closure. That is, we compute a *Datalog-rewriting of* Σ: a finite set of Datalog rules P_Σ such that for any base instance I, the ground closure of I with P_Σ is the same as the ground closure of I with Σ. Moving to a Datalog-rewriting reduces the fact entailment problem for Σ to the same problem for P_Σ. But to find the ground closure of I under P_Σ, we can use a standard Datalog engine. In particular the evaluation will terminate in time polynomial in |I| returning the ground closure.

Example 3.3. The two Datalog rules shown in Example 1.2 represent a rewriting of the TGDs in Example 1.1. By chasing with these rules on any source instance we will obtain all facts over base values that are entailed. ◁

The two-step approach to fact entailment given by Datalog rules not only separates concerns but gives procedures that scale well in the number of facts, allowing the use of database technologies [2, 32]. Thus, our goal is to *derive efficient Datalog-rewriting procedures for Guarded TGDs*

4 CHASING AND REWRITING

The One-Pass Property. In a general tree-like chase proof, the firing order of rules may not respect the tree structure. To get from the chase to rewriting rules, it will be useful to have a more restrictive notion of chase proofs that enforces a closer connection between the tree structure and firing order.

For any tree-like chase sequence, we can associate with each T_i the *recently-updated node*: the node that is modified, by being created or having its facts updated, in moving from T_{i-1} to T_i for $i \geq 1$. We say that a tree-like chase sequence $T_0, \ldots, T_n$ is *one-pass* if (1) every propagation step propagates facts from a child to a parent (2) we only make a chase step in T_i at an ancestor-or-self of the recently-updated node of T_i, (3) similarly, we only propagate a fact F out of a node v if v is the recently-updated node or an ancestor of that node. The name "one-pass" indicates that once we "leave" a subtree of a node v — by performing a propagation or chase step outside of it — we cannot return to it.

Example 4.1. The chase depicted in Figure 1 is *not* one-pass. From step 6 to 7, we update a node that is not an ancestor-or-self of the recently-updated node. We are propagating a fact to a "cousin" of the recently-updated node. In the next step, we make crucial use of this propagated fact.

In order to obtain a one-pass chase, instead of propagating a fact directly from a node v to a node v', we can first propagate the fact to a common ancestor of v and v' and then re-create v' from that ancestor. This idea is depicted in Figure 2, and it is the key idea of the proof of Theorem 4.2.

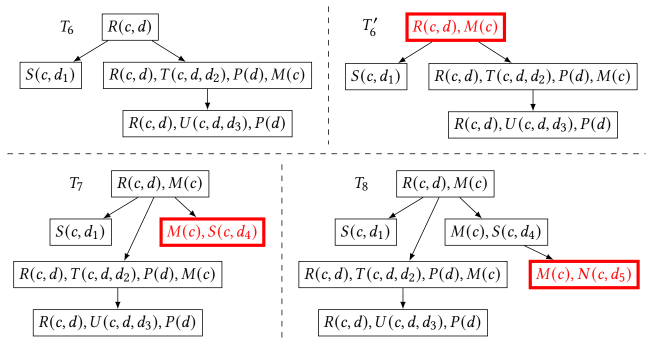


Figure 2: We can modify the chase from Figure 1 to obtain a one-pass chase. The recently-modified node is marked red.

We say that a chase sequence is *one-fact* if for each node $v \in T_i$ and child v' of v, there is at most one fact in the subtree of v' that is propagated to v.

We show that we can indeed assume the existence of a one-pass and one-fact chase proof whenever we have a chase proof for some *BCQ Q*. We achieve this by constructing a suitable one-pass, one-fact chase sequence from a given arbitrary tree-like chase sequence, as sketched in Example 4.1.

Theorem 4.2. *For every set of GTGDs in head normal form, for every tree-like chase sequence $T_0, \ldots, T_n$, there is a one-pass, one-fact chase sequence $\overline{T_0}(=T_0), \overline{T_1}, \ldots, \overline{T_m}$ such that there is a homomorphism $h : T_n \to \overline{T_m}$ with $h(c) = c$ for any constant c in T_0.*

In particular, a BCQ Q has a chase proof from I and GTGDs Σ if and only if it has a one-pass, one-fact chase proof from I and Σ.

Correctness criteria for a rewriting algorithm and the one-pass, one-fact chase. Say that a one-pass, one-fact chase sequence $T_0, \ldots, T_n$ for Σ is a *loop* (of Σ) if the step from T_0 to T_1 generates a new child c of some node p in T_0, each chase step from T_1 through T_{n-1} in the sequence impacts only the subtree of c, and the final chase step results in a single fact F being propagated to p. This is the *output fact of the loop*. Note that in any one-pass chase, in a loop that begins by generating c and ends with T_n, after T_n the subtree of c can never be modified. A *simple loop* is a loop where the tree structure of T_0 is a single node. A simple loop is thus a sequence that adds a single fact F to the root node of T_0.

For a set of derived Datalog rules Σ′ to be a rewriting of a set of TGDs Σ, we must ensure that whenever we have a node v in a chase tree and a fact F guarded by v is later created in the chase, then F can also be created by a sequence of Datalog rules using Σ′. It is evident that F cannot only be created by firing a Datalog rule in v but also by firing a rule in some descendant of v, as can be seen in Figure 1. But in the one-pass, one-fact chase, each new fact arrives by firing a Datalog rule from Σ in v or via one loop through a subtree generated from v. In either case, the witness sequence can be considered as a stand-alone proof with v as the root node. The following sufficient condition for correctness of a rewriting is thus easy to see:

Proposition 4.3. *Consider a set Σ′ of Datalog rules that are logical consequences of Σ. Suppose that for each simple loop $T_0, \ldots, T_n$ of Σ with output fact F, there is a TGD in Σ′ which when applied to T_0 produces F. Then Σ′ is a rewriting of Σ.*

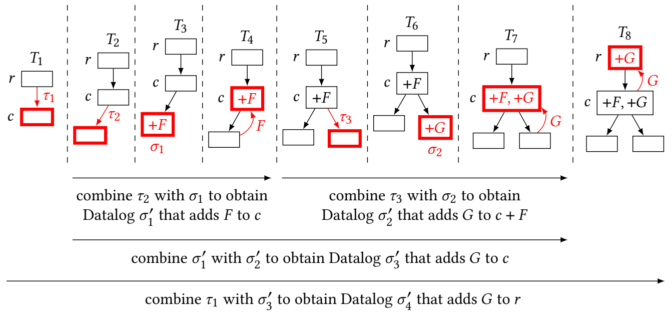
5 REWRITING ALGORITHMS IN LIGHT OF THE CHASE

We now consider different ways of obtaining a rewriting for a set of GTGDs, with the criteria of Proposition 4.3 in mind.

5.1 SimDR: creating Datalog rules inductively

Proposition 4.3 states that to get a Datalog-rewriting, it suffices to have rules that simulate simple loops in the one-pass chase. Our first algorithm is a naïve approach which will create *only Datalog rules (full TGDs)*, with the goal of fulfilling the criteria of Proposition 4.3.

Example 5.1. Consider a typical one-pass chase shown schematically in the top half of Figure 3. We apply a non-full rule τ_1 to the



T1
T2
T3
T4
T5
T6
T7
T8
r
c
τ1
τ2
τ3
+F
+G
+F, +G
F
G
σ1
σ2
combine τ2 with σ1 to obtain Datalog σ′1 that adds F to c
combine τ3 with σ2 to obtain Datalog σ′2 that adds G to c + F
combine σ′1 with σ′2 to obtain Datalog σ′3 that adds G to c
combine τ1 with σ′3 to obtain Datalog σ′4 that adds G to r

Figure 3: One-pass chase and forming a rewriting inductively for the example

root to create a child c, apply another non-full rule τ_2 to the child to create a grandchild of the root, then apply a Datalog rule σ_1 to the grandchild to create fact F that can be propagated to the child. Thus, T_2, T_3, T_4 is a loop, and by focusing on the subtree rooted at that c it is a simple loop, call it sl^c_1. We then repeat a similar process at the child, applying a non-full rule τ_3 that creates a second grandchild of the root, then a Datalog rule σ_2 that generates fact G which can be propagated back to the child. This creates a second loop, sl^c_2 at the child. Composing these loops we get a loop sl^c_3 at the child that adds G to the birth facts of c. The fact G is guarded by the root, so it can be propagated back to the root, thus completing a simple loop sl^r_1 at the root.

An obvious idea to create a rewriting is to work backwards from the leaves. The process is illustrated in the bottom half of Figure 3.

We should first create a Datalog rule σ'_1 that produces the output facts of sl^c_1: we should be able to do this by combining the two rules τ_2 and σ_1 that were used in this loop. Similarly, we could create a derived Datalog rule σ'_2 that simulates sl^c_2 by combining τ_3 and σ_2. Composing σ'_1 and σ'_2 we should be able to get a Datalog rule σ'_3 that simulates the simple loop sl^c_3. Now we can pretend that the chase was created by first applying τ_1 to the root and then applying σ'_3 to the child. Thus combining τ_1 and σ'_3 we should be able to get a single Datalog rule σ'_4 that can simulate the simple loop sl^r_1 when applied to the root.

Note that in performing this inductive process, we implicitly used the fact that the evolution of the subtree of c represent a self-contained one-pass chase proof.

Using these observations, we now present an algorithm that returns a Datalog-rewriting.

Definition 5.2 (Simple Datalog-rewriting, SimDR*).* Given a set of GTGDs Σ, the *Simple Datalog rewriting* $\mathsf{SimDR}(\Sigma)$ is defined via taking the set of Datalog rules within the closure of Σ under the following inference rules:

- (COMPOSE). We can resolve two Datalog TGDs:
  Assume we have two Datalog rules $\tau, \tau' \in \mathsf{SimDR}(\Sigma)$ of the form
  $$\beta(\vec{x}) \to \eta(\vec{x}),$$
  $$\beta'(\vec{z}) \to \eta'(\vec{z})$$
  that use distinct variables (otherwise, perform a renaming). Suppose there is a unifier θ of some atom in η and an atom in β' such that the number of variables in range θ has cardinality at most $\mathsf{hwidth}(\Sigma) + |\mathsf{consts}(\Sigma)|$. Then we add the Datalog $(\theta(\beta) \cup (\theta(\beta') \setminus \theta(\eta)) \to \theta(\eta'))$.
- (ORIGINAL). We can resolve a non-full with a Datalog TGD if this generates a Datalog rule:
  Assume we have a non-full $\tau \in \Sigma$ and a Datalog rule $\tau' \in \mathsf{SimDR}(\Sigma)$ of the form
  $$\beta(\vec{x}) \to \exists \vec{y}\, \eta(\vec{x}, \vec{y}),$$
  $$\beta'(\vec{z}) \to \eta'(\vec{z})$$
  and assume that τ, τ' use distinct variables (otherwise, perform a renaming). Suppose θ is an mgu of atoms $S := (H_1, \ldots, H_n)$ and $S' := (B'_1, \ldots, B'_n)$ that is the identity on $\vec{y}$,[1] where $H_i \in \eta$, $B'_i \in \beta'$ for $1 \leq i \leq n$. Moreover, assume that the following check is satisfied:

  Any atom $B' \in \beta'$ with $\mathsf{vars}(\theta(B')) \cap \vec{y} \neq \emptyset$ occurs in S', $\theta(\vec{x}) \cap \vec{y} = \emptyset$, and $\mathsf{vars}(\theta(\beta')) \subseteq \theta(\vec{x}) \cup \vec{y}$.

  Then, if η'' is non-empty, we add the Datalog rule $\beta'' \to \eta''$ where
  $$\beta'' := \theta(\beta \cup (\beta' \setminus \{B'_1, \ldots, B'_n\})),$$
  $$\eta'' := \{H' \in \theta(\eta') \mid \mathsf{vars}(H') \cap \vec{y} = \emptyset\}.$$

Normalization and termination. All of our rewriting algorithms will produce TGDs with a number of variables that can be bounded by a number w that can be derived from the input. We will implicitly normalize all intermediate outputs to use canonical variable names $x_1 \ldots x_w$, introduced in textual order. Our algorithm will always discard a rule that is a repeat, and this will ensure termination. In fact, in our implementation we will remove rules that are *subsumed* by an existing output: our subsumption techniques are discussed in Section 6.

Theorem 5.3. *The* SimDR *algorithm provides a valid rewriting, hence gives a decision procedure for fact entailment.*

The proof of completeness given in Theorem 5.3 is an induction on the size of a one-pass chase sequence witnessing the entailment of the output fact F in Proposition 4.3. The inductive construction generalizes the intuition of building up larger and larger rewritings from chase proofs from Example 5.1. We can also show (see the supplementary material) that the procedure terminates in PTIME for fixed Σ, EXPTIME for schemas of bounded arity, and in 2EXPTIME in general: this matches the tight lower bounds known for fact entailment [14].

The Simple Datalog-rewriting has several obvious weak points. It will generate Datalog rules for *all* composition of loops since it is closed under composition of Datalog rules. This is intuitively unnecessary. On the other hand, the rule (COMPOSE) cannot simply be dropped while maintaining completeness since in the (ORIGINAL)-step we may need to compose a non-full rule with a composition of two Datalog rules in order to create another Datalog rule. Moreover, in the (ORIGINAL)-step, it is not clear how one efficiently obtains

[1]Such an mgu can be obtained by treating $\vec{y}$ as constants.

the sequences S, S'. The following example illustrates another drawback of SimDR.

Example 5.4. Given the instance $I = \{R(c, d)\}$ and a set of GTGDs Σ

$$\tau_1 = R(x_1, x_2) \rightarrow \exists y_1, y_2 \big(S(x_1, x_2, y_1, y_2) \wedge T(x_1, x_2, y_2)\big),$$
$$\tau_2 = S(x_1, x_2, x_3, x_4) \rightarrow U(x_4),$$
$$\tau_3 = T(z_1, z_2, z_3) \wedge U(z_3) \rightarrow P(z_1),$$

it is clear that $I, \Sigma \models P(c)$. Consider the steps performed by SimDR. In the first iteration, (ORIGINAL) does not create any new rules since the composition of τ_1, τ_2 does not contain head atoms without existential variables and composing τ_1, τ_3 creates an existential variable in $U(z_3)$. We can apply the (COMPOSE)-rule with τ_2, τ_3. But it is not so clear which unifier θ, identifying variables z_i in τ_3 with variables x_i in τ_2, one should consider. One natural way is to restrict θ to be $\theta = [x_4/z_3]$, with the resulting resolvent corresponding to $S(x_1, x_2, x_3, x_4) \wedge T(z_1, z_2, x_4) \rightarrow P(z_1)$. But this rule contains more than $\mathsf{hwidth}(\Sigma) = 4$ variables and hence will not be derived by (COMPOSE). Eliminating the upper bound on the number of variables is no solution as this restriction is needed for termination purposes. Instead, every possible unifier that keeps this upper bound is considered, leading to a series of derived rules including

$$S(x_1, x_2, x_3, x_4) \wedge T(x_1, x_2, x_4) \rightarrow P(x_1)$$
$$S(x_1, x_2, x_3, x_4) \wedge T(x_2, x_1, x_4) \rightarrow P(x_2),$$
$$S(x_1, x_2, x_3, x_4) \wedge T(x_1, x_3, x_4) \rightarrow P(z_1),$$
$$S(x_1, x_2, x_3, x_4) \wedge T(x_3, x_1, x_4) \rightarrow P(x_3),$$

and so forth, which is infeasible in practice. ◁

We hence look for a more efficient rewriting algorithm.

5.2 The Existential-based Rewriting

Again, we describe the idea of the algorithm by example.

Example 5.5. Let us revisit the schematic tree-like chase from Example 5.1 and Figure 3. We can begin as before, creating a derived Datalog rule σ'_1 that simulates the simple loop sl^c_1 by combining τ_2 and σ_1 and a derived Datalog rule σ'_2 that simulates sl^c_2 by combining τ_3 and σ_2.

But now we can combine the non-full rule τ_1 with σ'_1 to create a *derived non-full rule* τ'_1 which simulates not a loop, but a "depth-one evolution": a series of chase steps that creates a child and then updates it. We can then combine τ'_1 with σ'_2 to simulate an even larger depth-one evolution, one which can be applied to the root to produce an updated child that includes facts F and G. This rule can then be normalized into two separate rules, which produces a full rule. The process is shown in Figure 4.

Remark 5.6. *The unification issue of Example 5.4 now disappears: we first compose τ_1, τ_2 to obtain the guarded*

$$\tau'_1 := R(x_1, x_2) \rightarrow \exists y_1, y_2 \big(S(x_1, x_2, y_1, y_2) \wedge T(x_1, x_2, y_2) \wedge U(y_2)\big)$$

and then compose τ'_1, τ_3 to obtain $\tau'_2 := R(x_1, x_2) \rightarrow P(x_1)$.

Using these observations, we now present our improved rewriting algorithm for GTGDs. The main idea is to drop (COMPOSE) from SimDR and allow (ORIGINAL) to derive non-full rules:

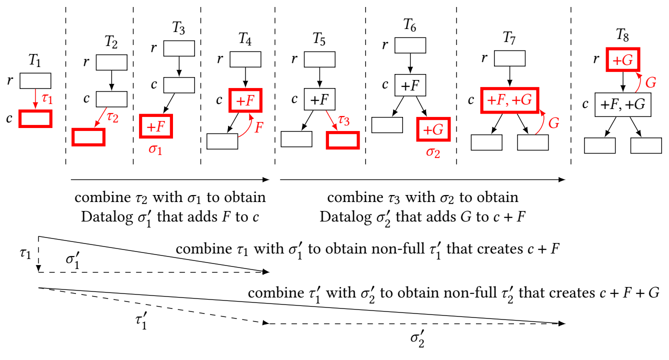

T_1
r
c
τ_1
T_2
τ_2
T_3
$+F$
σ_1
T_4
F
T_5
τ_3
T_6
$+G$
σ_2
T_7
$+F, +G$
G
T_8
combine τ_2 with σ_1 to obtain Datalog σ'_1 that adds F to c
combine τ_3 with σ_2 to obtain Datalog σ'_2 that adds G to $c + F$
combine τ_1 with σ'_1 to obtain non-full τ'_1 that creates $c + F$
σ'_1
combine τ'_1 with σ'_2 to obtain non-full τ'_2 that creates $c + F + G$
τ'_1
σ'_2

Figure 4: Alternative Rewriting for Example 5.1

Definition 5.7 (Existential-based rewriting, ExbDR*).* We define the following inference rule:

- (G-RESOLVE). Assume we have a non-full τ and a Datalog rule τ' of the form

$$\beta(\vec{x}) \rightarrow \exists \vec{y}\, \eta(\vec{x}, \vec{y}),$$
$$\beta'(\vec{z}) \rightarrow \eta'(\vec{z}),$$

and assume that τ, τ' use distinct variables (otherwise, perform a renaming). Suppose θ is an mgu of atoms $S := (H_1, \ldots, H_n)$ and $S' := (B'_1, \ldots, B'_n)$ that is the identity on $\vec{y}$, where $H_i \in \eta$, $B'_i \in \beta'$ for $1 \leq i \leq n$. Moreover, assume that the following existential variable check is satisfied:

Each atom $B' \in \beta'$ with $\mathsf{vars}(\theta(B')) \cap \vec{y} \neq \emptyset$ occurs in S' and $\theta(\vec{x}) \cap \vec{y} = \emptyset$.

We then add $\beta'' \rightarrow \eta''$ where

$$\beta'' := \theta\big(\beta \cup (\beta' \setminus \{B'_1, \ldots, B'_n\})\big),$$
$$\eta'' := \theta(\eta \cup \eta').$$

Given a set of GTGDs Σ, consider the closure of Σ under rounds consisting of firing the inference rule above, at each step converting any newly-generated constraint to head normal form, normalizing the variables, and removing duplicates as usual: we denote the fixpoint of this process as the *intermediate existential-based rewriting* $\mathsf{IExbDR}(\Sigma)$ while the *output existential-based rewriting* $\mathsf{ExbDR}(\Sigma)$ is obtained from $\mathsf{IExbDR}(\Sigma)$ by dropping all those intermediate non-full rules, keeping only the Datalog rules.

We can show that $\mathsf{ExbDR}(\Sigma)$ satisfies the completeness criterion of Proposition 4.3. The argument generalizes the inductive construction of Example 5.5 that simultaneously creates non-full rules from depth-one evolutions in the one-pass chase and Datalog rules from simple loops. Thus we have:

Theorem 5.8. $\mathsf{ExbDR}(\Sigma)$ *is a Datalog-rewriting of* Σ.

We can also show that any rule in $\mathsf{IExbDR}(\Sigma)$ is guarded. This is unlike SimDR which might produce unguarded rules due to (COMPOSE) (cf. Example 5.4). It then turns out that in the (G-RESOLVE) rule, once we have fixed a unifier with a guard atom, we know everything we need to know about S'.

Proposition 5.9. *Assume we perform a* (G-RESOLVE) *inference on τ, τ' and θ^* is the mgu of a head atom of τ with the guard atom* *of τ' that is the identity on $\vec{y}$. Then B' occurs in S' if and only if* $\mathsf{vars}(\theta^*(B')) \cap \vec{y} \neq \emptyset$.

Thus the main choices we have are the choice of TGDs to (G-RESOLVE), the head atom to unify with the guard, and the remaining head atoms in the other rule that can be unified: the set S. Since the unification of the head atom usually heavily constraints the remaining possibilities, the number of options to explore is usually very small. For example, in our experiments we find that in 98% of the calls to (G-RESOLVE) there is only one choice, and the maximum number of choices overall occurring in our dataset is 3. We discuss implementation details in the supplementary material.

5.3 Using Skolem functions: SkolemDR

The previous algorithm produced non-full TGDs which may have multiple existentially-quantified atoms in the head. In fact, the heads of these TGDs can grow significantly during runs of the algorithm.

Example 5.10. For any n, consider the theory Σ_n consisting of the single non-full GTGD

$$A(x) \rightarrow \exists y\, B(x,y)$$

along with the Guarded Datalog rules

$$B(x,y) \wedge C_i(x) \rightarrow D_i(y)$$

for $1 \leq i \leq n$.

ExbDR will produce a collection of non-full TGDs that grows exponentially in n. For I a subset of $\{1,\dots,n\}$ it will produce a TGD R_I of the form

$$A(x) \wedge \bigwedge_{i\in I} C_i(x) \rightarrow \exists y\, B(x,y) \wedge \bigwedge_{i\in I} D_i(y).$$

In Section 6 we will introduce optimizations, including one related to subsumption of rules. But this will not diminish the blow-up here since R_I is not subsumed by R_J for any $I \neq J$. ◁

Intuitively, the blow-up in the example above is unnecessary. If we *Skolemize* the non-full TGD, which means giving a name $f(x)$ to the y that is created for a given x in the first rule, then it should be possible to only derive the rules

$$A(x) \wedge C_i(x) \rightarrow D_i\big(f(x)\big)$$

for $1 \leq i \leq n$. Here f is a new function symbol.

Above we are presenting an example of a *Skolemized TGD*, defined like a TGD but where β and η use atoms that may contain function symbols, but no existential quantifiers. We will assume that our Skolemized TGDs have only a single atom in the head; this can always be achieved by splitting up into multiple rules. We say that an atom is *functional* if it contains a function symbol; otherwise it is said to be *function-free*. A Skolemized TGD is function-free if all atoms in it are function-free. If we have an ordinary dependency $\tau = \forall\vec{x}\ [\beta \rightarrow \exists\vec{y}\ \eta]$ with $\vec{x}'$ the exported variables, we can *Skolemize* τ to get the corresponding Skolemized TGD

$$\forall\vec{x}\ [\beta(\vec{x}) \rightarrow \sigma(\eta)].$$

where σ is a substitution mapping each variable y_i to a term $f_i(\vec{x}')$ with f_i a fresh function symbol.

Note that TGDs and Skolemized TGDs are formally incomparable. The former may have existentials in the head, but not Skolem functions. The latter may have functions in the heads and bodies, but not existentials in the head. We will use the term *rule* to refer to either a TGD or Skolemized TGD in contexts where the distinction between the two is unimportant.

We can check the intuition that the blow-up in the example above is avoided by the Skolemized variant of ExbDR below:

Definition 5.11 (Skolem rewriting, SkolemDR*).* We define the following inference rule on single-headed guarded rules:

- (SK-RESOLVE). Assume we have some $\tau = \beta \rightarrow H$ with a single functional atom H in the head and β function-free, a rule $\tau' = \beta' \rightarrow H'$, and some $B' \in \beta'$ such that
  - τ' is function-free and B' is a guard, or
  - B' is functional,

  and assume that τ, τ' use distinct variables (otherwise, perform a renaming). Suppose θ is an mgu of H and B'. We then add $\beta'' \rightarrow H''$ where

$$\beta'' := \theta\big(\beta \cup \big(\beta' \setminus \{B'\}\big)\big),$$
$$H'' := \theta(H').$$

The *intermediate Skolem rewriting* (ISkolemDR) is defined by iterating the above inference, performing renamings of variables to get canonical variable names as usual until reaching a fixed point. The *output Skolem rewriting* (or simply "Skolem rewriting" when clear from context) consists of the function-free rules in the intermediate Skolem rewriting.

The intuition is that we can resolve a τ that has no Skolem functions in the body with a τ' that is in Datalog. This is the first case above, and it is analogous to the rule of ExbDR which resolves a non-full and a Datalog rule. But now the resulting rule can have Skolem functions in the body as well as the head, and we need a way to remove those Skolem functions. The second case above gives us a means to eliminate one of these atoms in the body (B').

Example 5.12. We return to Example 5.10 and consider how SkolemDR behaves. It will first produce

$$A(x) \rightarrow B(x, f(x)).$$

to obtain a functional head atom. It then continues to create the non-full TGDs

$$A(x) \wedge C_i(x) \rightarrow D_i\big(f(x)\big)$$

for $1 \leq i \leq n$, as we desired. ◁

Example 5.10 and 5.12 show that SkolemDR can perform exponentially better than ExbDR. We can also show (see the supplementary material) that *the size of* ISkolemDR *is always within a polynomial of the size of* IExbDR. Thus, *the opposite kind of example is impossible.* Again, we can use Proposition 4.3 to show that SkolemDR is complete:

Theorem 5.13. *The output Skolem rewriting* SkolemDR(Σ) *is a rewriting of* Σ.

5.4 Looking ahead: rewriting based on hyper-resolution

Recall that in SkolemDR we create rules with Skolem terms in the head and the body. We call a rule *body-full* if it has no Skolems in the body, and *body-non-full* otherwise. Body-full rules are the analogues of the rules produced by ExbDR — either Datalog rules, or rules with function symbols only in the head. Note that rules with Skolem terms in the body are useful only as intermediate steps to eventually derive new Datalog rules. A concern about SkolemDR is that we may spend time creating body-non-full rules that never produce any body-full ones.

Example 5.14. Consider how SkolemDR behaves on the following GTGDs:

$$\begin{aligned}\tau &= S(x) \to \exists y\, R(x,y),\\ \tau_i &= R(x,y) \wedge A_i(x) \to B_i(y), \text{ for } 1 \le i \le n,\\ \tau' &= B_1(x) \wedge B_2(x) \wedge \cdots \wedge B_n(x) \to G(x).\end{aligned}$$

At the beginning, we Skolemize τ, and then we can resolve it with each τ_i, returning the rules $\tau_i' := S(x) \wedge A_i(x) \to B_i\big(f(x)\big)$. Then, we can resolve any τ_i' with τ'; for example resolving τ_1' with τ' produces

$$S(x) \wedge A_1(x) \wedge B_2\big(f(x)\big), \ldots, B_n\big(f(x)\big) \to G\big(f(x)\big).$$

This can be generalized by considering a non-empty subset I of $\{1,\ldots,n\}$ and τ_I' being the resulting evolution of τ_i', for $i \in I$, with τ', giving

$$\tau_I' := S(x) \wedge \bigwedge_{i\in I} A_i(x) \wedge \bigwedge_{j\in\big(\{1,\ldots,n\}\setminus I\big)} B_j\big(f(x)\big) \to G\big(f(x)\big).$$

Thus, SkolemDR will produce an exponential output, even considering subsumption. Most of these outputs do not yield any interesting Skolem-free GTGDs. ◁

One way to address this is to avoid generating body-non-full rules completely. We do this by applying *hyper-resolution* [37], in which multiple clauses are composed with a single clause.

Definition 5.15 (Hyper-Resolution Rewriting, HyperDR*).* We define the following hyper-resolution inference rule:

- (G-HYP). Assume we have body-full but non-full rules $\tau_i = \beta_i \to H_i$, for $1 \le i \le n$, and one Datalog rule $\tau' = \beta' \to H'$ with distinct variables. Suppose θ is an mgu of $(H_1,\ldots,H_n)$ with some $(B_1',\ldots,B_n')$ where $B_i' \in \beta'$. Then, if β'' is Skolem-free, we add $\beta'' \to H''$ where

$$\begin{aligned}\beta'' &:= \bigcup_{1\le i\le n} \theta(\beta_i) \cup \theta\big(\beta' \setminus \{B_1',\ldots,B_n'\}\big),\\ H'' &:= \theta(H').\end{aligned}$$

The *intermediate hyper-resolution rewriting* (IHyperDR) is the closure under this inference rule, normalizing and removing duplicates as usual. The restriction of the result to function-free rules is referred to as the *output hyper-resolution rewriting* (of Σ) or simply the hyper-resolution rewriting of Σ, HyperDR.

Example 5.16. We return to Example 5.14. In HyperDR, we obtain τ_i', for $1 \le i \le n$, as above. But then we search for a hyperresolvent, which will require matching each τ_i''s head with the corresponding B_i atom in τ''s body. Thus, a single hyperresolution step will directly give

$$S(x) \wedge \bigwedge_{1\le i\le n} A_i(x) \to G\big(f(x)\big).$$

◁

We can give an argument for completeness of HyperDR by simulation of ExbDR.

Theorem 5.17. *HyperDR(Σ) is a Datalog-rewriting of Σ.*

The drawback of HyperDR is that determining the possible hyperresolutions can be expensive, since it involves determining whether there is a match involving a combination of multiple rules. In our implementation we first choose τ_i unifying with the guard of the Datalog rule τ, where the unification may introduce skolems. We find, for each atom with a skolem function in the unified body of τ, the set C_j of τ_j whose head can provide that body atom, possibly with some further unification. We then enumerate elements in the product of the C_j to see if they give a valid HyperDR step. When one of the C_j is empty or many of the C_j have only one element, exploring the product can be done quickly, and HyperDR provides an enormous benefit. This is the case in Example 5.16.

6 IMPLEMENTING REWRITING

Up until now we have described rewriting algorithms as taking a fixpoint of some rule closure process, and then restricting to the Datalog rules. Having presented a number of such closure processes and argued that they give rewritings, we now talk about the algorithms and their implementation in more detail. We will employ techniques that are applicable to all the algorithms in the previous section, each technique being a variation of those introduced for rewriting algorithms for description logics [32]. They include *subsumption* and *unification indexing*, as well as *structure-preserving transformation.*

The generic top-level loop used in the implementation of each algorithm is given in Algorithm 1. A difference occurs for HyperDR at line 13, where instead of a single appropriate $\tau \in \mathcal{D} \cup \mathcal{N}$, a set of such TGDs is needed to apply the hyper-resolution. To ensure termination we need to normalize rules and at the very least remove syntactical duplicates. To terminate more quickly, we prefer to remove rules if they are *subsumed* – that is, are logically weaker than – another rule. Consider a TGD $\tau = \beta(\vec{x}) \to \exists\vec{y}\ \eta(\vec{x},\vec{y})$ and another TGD $\tau' = \beta'(\vec{x}') \to \exists\vec{y}'\ \eta(\vec{x}',\vec{y}')$. Subsumption of τ by τ' means that β is at least as strong (and possibly stronger than) β', while η is no stronger than η'. As indicated in Algorithm 1, before adding a newly derived TGD τ to the working set $\mathcal{W}$, we do two kinds of subsumption steps. First there is a *forward subsumption* step i.e. we check if τ is subsumed by any τ' in the set of current intermediate TGDs $\mathcal{I} = \mathcal{W} \cup \mathcal{N} \cup \mathcal{D}$, and in this case, we prevent τ to be added to $\mathcal{W}$. Second comes a *backward subsumption* step, i.e. we gather all the TGDs τ' in $\mathcal{I}$ subsumed by τ and we remove them. The same idea applies to Skolemized TGDs in the context of SkolemDR or HyperDR.

Since an exact test for logical containment is expensive, we use two ideas from the literature [39] to make these checks efficient. The first idea is to find an approximation that is easy to check. A *subsumption necessary condition* is a Boolean function N on pairs (τ,τ') such that if τ is logically stronger than τ', then $N(\tau,\tau') =$ True; that is, it overapproximates subsumption. A *subsumption sufficient condition* is a Boolean function S on pairs (τ,τ') such that $S(\tau,\tau') =$ True implies that τ is logically stronger than τ'. An example of a subsumption sufficient condition is when τ' is *syntactically subsumed by* τ: β' contains all atoms of β and η contains all atoms of η'. An

Algorithm 1: Top-level loop of rewriting algorithms.

```
Function Top − levelRewrite:
    Input  : Guarded Datalog and non-full TGDs $\Sigma_d \cup \Sigma_n$
    Output : Rewriting of $\Sigma$
    $\mathcal{W} = \Sigma_n$            // working set
    $\mathcal{D} = \Sigma_d$            // Datalog set
    $\mathcal{N} = \emptyset$            // non-full set
    while $\mathcal{W} \neq \emptyset$ do
        Let $\sigma$ be a rule in $\mathcal{W}$
        $\mathcal{W} = \mathcal{W} \setminus \{\sigma\}$
        if $\sigma$ is Datalog then
            $\mathcal{D} = \mathcal{D} \cup \{\sigma\}$
        else
            $\mathcal{N} = \mathcal{N} \cup \{\sigma\}$
        end
        $\mathcal{E} =$ apply inference rules to $\sigma$ and appropriate $\tau \in$
          $\mathcal{D} \cup \mathcal{N}$
        for $\rho \in \mathcal{E}$ do
            if $\rho$ is not a syntactic tautology and $\rho$ is not
              (forward) subsumed in $\mathcal{W} \cup \mathcal{D} \cup \mathcal{N}$ then
                Remove from $\mathcal{W}, \mathcal{D}, \mathcal{N}$ the TGDs subsumed
                  by $\rho$      // backward subsumption
                $\mathcal{W} = \mathcal{W} \cup \{\rho\}$
            end
        end
    end
    return $\mathcal{D}$
```

example of a subsumption necessary condition is that τ' *relation subsumes* τ: the relations in β are contained in those in β', and the relations in η' contained in those of η. Our approach will be to first apply a subsumption necessary condition: if the condition fails we know that τ' is not subsumed. We can then apply a subsumption sufficient condition on the remaining pairs to decide whether τ' is subsumed by τ.

The second idea is to use a tree index to retrieve subsumption candidates. We use a set of objects called features and a function efficiently associating each TGD with two subsets of the features, the positive features denoted by F^+ (resp. the negative features denoted by F^-), which only depends on the head (resp. body) of the TGD. We enforce that each feature f has the following properties: If τ subsumes τ' then (1) if f is positive and τ' has f, then τ has f, and (2) if f is negative and τ has f, then τ' has f. Hence, given a set of features F, we define our necessary condition by $N(\tau, \tau') = \text{True}$ if and only if $F^+(\tau') \subseteq F^+(\tau)$ and $F^-(\tau) \subseteq F^-(\tau')$.

We explain the idea of features with a simple instantiation of it. We fix an ordering of pairs $P_i = (R_i \in \mathit{Preds}, p_i \in \{+,-\})$, where Preds is the set of predicates. The *feature vector* for τ is a list of booleans $F_i(\tau)$ where $F_i(\tau)$ is true if p_i is $+$ and τ contains R_i in the body, or p_i is $-$ and τ contains R_i in the head.

Example 6.1.[TGDs features] Consider the following TGDs:

$$\tau_1 := A(x) \to \exists y\, B(x,y) \wedge C(y),$$
$$\tau_2 := A(x) \wedge C(x) \to B(x,x)$$

And assume our ordering is $(A,-)$, $(B,-)$, $(C,-)$, $(A,+)$, $(B,+)$, $(C,+)$. Then the feature vector of τ_1 is (False, True, True, True, False, False), and the feature vector of τ_2 is (False, True, False, True, False, True). ◁

Our index will be a decision tree with the i^{th} branching being based on whether P_i is true. Thus in the example, the root will have two children, one leading to TGDs where A is in the head and the other one leading to TGDs where A is not in the head. We prune the tree to keep only the nodes that have non-empty children below them. Given such a tree, we can easily look up a set of leaves representing candidates that may be subsumed by τ:

For any decision representing a positive feature $(R_i,+)$, if R_i is in the body of τ then a candidate for subsumption by τ should contain R_i in the body, so we follow the edge corresponding to "true". If R_i is not in the body of τ then the feature does not constrain the candidate at all, so we follow both edges. The logic is reversed for negative features. Finishing this traversal for the TGD τ_1 in the example, we end up with a set of leaves containing exactly those TGDs whose bodies contain some atom with predicate A and whose heads do not contain any atoms with predicates other than B or C. These are exactly the TGDs τ' satisfying $N(\tau', \tau)$. We can then check for each τ' if it is actually subsumed by τ. The same idea can dually be applied to find the candidates that possibly subsume τ.

A remaining issue is that the ordering of the features impacts the performance of finding the candidates. We decided to use the ordering where negative features precede positive ones, and both positive and negative features are separately ordered by their frequency in the input TGDs.

We realized that, on the one hand, using the set of features defined above hurts performance when looking for candidates because the index becomes large; but on the other hand, the number of found candidates is small. To improve the overall performance of subsumption checking, we chose a smaller set of features whose corresponding index retrieves more candidates in a smaller time. The goal is to balance the time spent for finding the candidates and the time spent checking for syntactical subsumption between τ and the found candidates.

We form two partitions C^- and C^+ of the predicates and for each cluster of symbols c in one of these partitions, we create a feature f_c^- or f_c^+. A TGD has the feature f_c^- (or f_c^+), if its body (or its head) contains at least one symbol in c. The index based on these features is a simple generalization of the one for individual predicates. We selected our partitions C^- and C^+ with a fixed number of partition elements such that the number of input TGDs having the feature f_c^- is similar for each partition, and similarly for C^+. The number of clusters is computed using the average numbers of symbols and of atoms in the input TGDs. The aim of this symbol clustering is to obtain a balanced subsumption index, that is, an index in which the number of TGDs stored in each leaf is balanced.

Unification indexing. A unification index is used to filter the TGDs containing an atom that may be unifiable with a wanted atom. Two indexes are in use depending on whether we are looking for unifiable atoms appearing in a TGDs body or head. These indexes are used at Line 13 of Algorithm 1 to quickly find the pairs (or sets in HyperDR) of TGDs on which an inference rule could be applied. Our indexing is a simplified variant of the *path indexing* technique [40]. In the absence of Skolem functions and constants, each index boils down to a map associating to each relation the set of TGDs that use this relation in their body (or their head). With functions, the index is refined to prune from the selected TGDs those whose unifications between the wanted atom and a TGD atom having the same relation raises a clash due to a function mismatch.
Cheap lookahead optimization. Recall that HyperDR can be seen as using a look ahead to ensure that a series of resolution steps eventually produces something relevant; however, this look ahead is expensive. In SkolemDR and ExbDR we do a much simpler and cheaper optimization in the same spirit: we discard every non-full TGD τ_1 derived by a resolution involving the non-full τ if the atoms that are in τ_1's head but not in τ's head use predicates that are not in the body of any input TGDs. This is called *cheap look ahead*. In Example 5.10, the predicates D_i are introduced in the head of derived non-full TGDs, but there is no initial TGD using a predicate D_i in its body; so ExbDR exponential derivation could have been avoided using this optimization. Despite this optimization, new input TGDs with each D_i in their body can be added to the example such that the exponential derivation still occurs.
Structural transformation. One technique used heavily in the context of rewriting for description logics is *structure-preserving transformations* [36]. For our purposes the key idea there is to introduce intermediate predicates for subformulas. In the context of ontologies, natural opportunities arise based on description logic constraints that mention "complex concepts" C – that is concepts C defined by a description logic expression ϕ_C. A structure-preserving transformation introduces a new predicate P_C for C, and any constraint mentioning C would give rise to a new logical expression containing P_C. For example in the setting of Horn DLs (as in [32]), this would be a specialized TGD with P_C in the head and/or the body. An additional axiom would be generated that guarantees that P_C is equal to its definition ϕ_C. If C is used frequently in the source axioms, then the resulting TGDs may be considerably more succinct than one would obtain from inlining C in each axiom: the gain is similar to the one we would get from using common subqueries in databases. Description-logic based systems like KAON2 [33] implement this directly on the source ontology. It is not clear how to generalize it effectively working directly on higher-arity GTGDs. However, when working on ontologies, we can make use of this transformation as a pre-processing step.

7 BENCHMARK AND EXPERIMENTS

The goal of the experiments is to compare the performance and the output size of introduced algorithms. On description logic ontologies, we compare these algorithms with a competitor KAON2. We will also observe the impact of the predicate arity on the performance. Details can be found in [11].

Datasets. Our first set of experiments uses GTGDs taken from ontologies. We use the Oxford ontology library [34]. These are all on schemas of arity two, but they include many features beyond Guarded TGDs– e.g. disjunction in the head and cardinality constraints. We made use of the Graal parser from the GraphIK team [7] to convert the ontologies, as OWL axiom sets, into logical form, and then discarded those that were not GTGDs. In order to compare our algorithms to KAON2, which does not support values contained in OWL axioms, we generated two versions of the GTGD sets: one with the GTGDs containing values, another one without. The number of axioms in each ontology and the number of GTGDs obtained after the transformation are similar. We conducted our experiments using 428 ontologies, which, once transformed, contained from 3 to 230K TGDs with on average more than 11K Datalog rules and 5K non-full TGDs. This number is quite high because a few ontologies are very large, but the median number of Datalog rules is 789 and the median number of non-full TGDs is 283.

In order to understand the impact of increasing arity on our algorithms, we devised an algorithm to "blow-up" the relations in the 428 sets of GTGDs from the ontology repository without values from arity two to higher arity: The method is parameterized by a blow-up factor b. We begin with a straightforward mapping that replaces individual positions of a predicate with b positions. In isolation, such a mapping does not impact the complexity of query answering problems since every step with the blown-up TGDs can be mimicked in the original arity two TGDs. The advantage of blowing up is that we can add additional side atoms to the guard, and can introduce permutations of the variables within a TGD.
Competitors. We compare the four versions of atomic rewriting introduced previously: SimDR, ExbDR, SkolemDR, and HyperDR. All of these make use of the subsumption techniques and unification indexing discussed in Section 6. We know of no competitor to our system for general Guarded TGDs. The other higher-arity reasoners we found supported either radically different subsets than GTGDs, and also focused on goal-driven query answering rather than saturation [3, 13, 43]. So for an external competitor we compared to the system KAON2 [25, 32, 33] on ontologies that come from description logics. KAON2 implements a variation of SkolemDR, also applying subsumption, unification, and structure-preserving transformation. In the KAON2 comparison, we compare only on the constraints supported by both our system and KAON2.
System comparison on ontologies. We experimented on the ontologies described earlier to compare the performance of our four algorithms and KAON2. The results are shown in Table 1.

In the experiments there were 32 ontologies containing more than 20K TGDs, and all the algorithms reach the timeout of 10min on these. For simplicity, these ontologies are ignored in this analysis. We also omitted SimDR in the tables. As expected, SimDR is the slowest algorithm: it reaches the timeout on 173 ontologies and there are only 3 ontologies where some algorithms reach the timeout but SimDR does not. For this reason, we do not consider SimDR in any of the remaining discussion.
Overall performance. We see a great diversity in the performance of the algorithms: ExbDR differs from SkolemDR and KAON2 on the algorithms where they reach the timeout together. As expected, there are ontologies where ExbDR suffers considerably against the others, reflecting the problem in Example 5.10. In fact, ExbDR

Table 1: At the top, the table about ontologies, at the bottom, the table about higher-arity GTGDs. On the left, the number of inputs on which the algorithm on the row wins against the one on the column by one order of magnitude; on the right, the number of ontologies on which two algorithms reach the timeout simultaneously

	Exb	Sk.	Hyper	KAON2	Exb	Sk.	Hyper	KAON2
Exb	0	19	0	19	29			
Sk.	37	0	0	26	1	19		
Hyper	37	12	0	31	3	11	14	
KAON2	35	15	0	0	5	15	14	34

	Exb	Sk.	Hyper	Exb	Sk.	Hyper
Exb	0	61	87	26		
Sk.	11	0	21	0	62	
Hyper	6	4	0	20	56	101

loses by two orders of magnitudes against another algorithm on 27 ontologies. But there are other examples where ExbDR can benefit from keeping rule heads together. Although we can show that this benefit can only be polynomial, it can still be significant (see the supplementary material for details). The overall performance of HyperDR stands out: it is faster than another algorithm by an order of magnitude in 76 ontologies and it is not slower by an order of magnitude than any other algorithm.

Impact of clustering in indexing. We observed that on many ontologies, clustering the nodes in the subsumption index reduces the subsumption checking performance. For example, the most impacted algorithm SkolemDR wins by one order of magnitude against KAON2 on only one ontology (instead of 26), when the clustering of the index is disabled. The fact that KAON2 does not cluster its subsumption index is a major contributor to the improved performance of SkolemDR against KAON2.

Impact of structural transformation. We also tested pre-processing rules using the structure-preserving transformation of KAON2 to the ontologies to create the input of SkolemDR and HyperDR; the transformation is less obviously applicable to ExbDR since the transformation deals with Skolemized rules. We observed that applying it helps SkolemDR, reducing one order of magnitude on 22 ontologies, and it does not hurt HyperDR's performance. While the transformation can thus be useful, it is less clear how to best adapt it to higher-arity TGDs that do not come from ontologies.

Quality of output. We check whether the Datalog rules outputted by the different algorithms are usable for query answering. The rewritings contain a median of 439 rules, and 878 for KAON2, which is higher since new rules are introduced by the structural transformation. Among the ontologies with a non-trivial Datalog rewriting, the number of derived rules is on average 2396, with a median of 98 and a maximum of less than 24K for every of our algorithm. Overall the maximal Datalog rewriting contains less than 69K rules. In the output of every algorithm, the rules are quite simple: the maximum number of body atoms is 4. Although there is no sample data for most examples, we created a small synthetic dataset for each of them (a variant of the “critical instance” where there is an element connected by many relations), and found that all the output rulesets were runnable in the engine RDFox.

System comparison on higher-arity GTGDs. We used the blown-up ontologies to analyze the performance of the algorithms on higher-arity TGDs. During this experiment the four algorithms reach the timeout together on 128 inputs. Ignoring these inputs, the results are shown in Table 1 at the bottom. We observe that ExbDR gains an advantage on these higher-arity GTGDs over the two other Skolem-based algorithms. ExbDR loses by an order of magnitude in 17 cases whereas SkolemDR and HyperDR lose on 65 and 108, respectively. Moreover, ExbDR reaches the timeout on significantly fewer ontologies. We found that in many ontologies in the repository, SkolemDR produces a large number of intermediate TGDs with Skolems in the body which do not eventually contribute to a Datalog rule. In HyperDR these intermediate TGDs are not generated, but the candidates that would produce them must be examined in order to be discarded, thus leading to a similar delay. We can prove (see the supplementary materials) that this gap between SkolemDR/HyperDR and ExbDR is at most polynomial unlike the potential gap in the other direction (see Example 5.10). However, in practice, this polynomial gap can be very significant.

Impact of subsumption and unification indexing. To see the impact of subsumption and unification indexing, we conducted experiments on the ontologies where we disabled either of both mechanisms. Disabling subsumption means that the forward subsumption of Algorithm 1 is replaced by checking whether ρ is not in $\mathcal{W} \cup \mathcal{D} \cup \mathcal{N}$ and removing the backward subsumption at the next line. When disabling subsumption, the performance of ExbDR and HyperDR are heavily negatively impacted: there are respectively 72 and 17 new ontologies on which they timeout. On the contrary, disabling subsumption has a positive impact on SkolemDR's performance in many cases, with a gain of one order of magnitude on 12 ontologies. The number of intermediate TGDs explodes in every algorithm with an average multiplicative factor of 104 for ExbDR, 185 for SkolemDR and 103 for HyperDR. It shows that in SkolemDR the time to check the subsumption is so important, that dealing with the explosion of redundant intermediate TGDs can lead to good performance, but doing so in ExbDR and HyperDR is not viable. Finally, we observed that using a unification index is essential to obtain good performance.

Comparison with alternative end-to-end approaches. Finally, we discuss the impact of our algorithm on end-to-end reasoning tasks, compared to the major alternative method proposed in the literature, the automata-based approach. We take the following example, a variant of one considered in [14]:

$$\begin{aligned}
\sigma_1 &\coloneqq A(x) \rightarrow \exists y (L(x,y) \wedge A(y)) \\
\sigma_2 &\coloneqq A(x) \rightarrow \exists y (R(x,y) \wedge A(y)) \\
\sigma_3 &\coloneqq L(x,y) \wedge A(y) \rightarrow B(x)
\end{aligned}$$

While using our Datalog rewriting, query answering required less than 200 milliseconds even on a dataset of 1M rows. In contrast, the automata-based approach requires analyzing an automaton over an alphabet with 2^{36} labels even on a dataset with two elements!

8 CONCLUSION

In this work we took a first step in lifting rewriting algorithms that had been developed for description logics to the setting of higher-arity TGDs, and seeing how the various approaches to lifting work in practice. In the process we have shown a close connection between saturation algorithms and normalized versions of the chase.

Our current work handles only Guarded TGDs and atomic queries. But our chase-based analysis can be applied to wider classes like frontier-guarded TGDs and conjunctive queries. We are exploring these extensions in ongoing work. A variation of the normalized chase and saturation approaches for Disjunctive Guarded TGDs has been outlined in the masters thesis of one of the authors (see [27]). We are also exploring how the implementation ideas outlined here apply to that setting.

REFERENCES

[1] 1990. The monadic second-order logic of graphs. I. recognizable sets of finite graphs. *Information and Computation* 85, 1 (1990), 12–75.

[2] Shqiponja Ahmetaj, Magdalena Ortiz, and Mantas Simkus. 2018. Rewriting Guarded Existential Rules into Small Datalog Programs. In *ICDT*.

[3] Mario Alviano, Nicola Leone, Marco Manna, Giorgio Terracina, and Pierfrancesco Veltri. 2012. Magic-Sets for Datalog with Existential Quantifiers. In *Datalog 2.0*.

[4] Antoine Amarilli and Michael Benedikt. 2018. When Can We Answer Queries Using Result-Bounded Data Interfaces?. In *PODS*.

[5] Hajnal Andréka, Johan van Benthem, and István Németi. 1998. Modal languages and bounded fragments of predicate logic. *Journal of Philosophical Logic* 27 (1998), 217–274.

[6] Franz Baader, Diego Calvanese, Deborah McGuinness, Peter Patel-Schneider, and Daniele Nardi. 2003. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge university press.

[7] J.-F. Baget, M. Leclère, M.-L. Mugnier, S. Rocher, and C. Sipieter. 2015. Graal: A Toolkit for Query Answering with Existential Rules. In *RuleML*.

[8] Jean-François Baget, Marie-Laure Mugnier, Sebastian Rudolph, and Michaël Thomazo. 2011. Walking the complexity lines for generalized guarded existential rules. In *IJCAI*.

[9] Vince Bárány, Michael Benedikt, and Balder Ten Cate. 2013. Rewriting Guarded Negation Queries. In *MFCS*.

[10] Luigi Bellomarini, Emanuel Sallinger, and Georg Gottlob. 2018. The Vadalog System: Datalog-based Reasoning for Knowledge Graphs. *Proc. VLDB Endow.* 11, 9 (2018), 975–987.

[11] Michael Benedikt, Maxime Buron, Stefano Germano, Kevin Kappelmann, and Boris Motik. 2021. Guarded Saturation repository on GitHub. https://krr-oxford.github.io/Guarded-saturation/.

[12] Michael Benedikt, George Konstantinidis, Giansalvatore Mecca, Boris Motik, Paolo Papotti, Donatello Santoro, and Efthymia Tsamoura. 2017. Benchmarking the Chase. In *PODS*.

[13] Michael Benedikt, Boris Motik, and Efthymia Tsamoura. 2018. Goal-Driven Query Answering for Existential Rules With Equality. In *AAAI*.

[14] Andrea Calì, Georg Gottlob, and Michael Kifer. 2013. Taming the Infinite Chase: Query Answering under Expressive Relational Constraints. *Journal of Artificial Intelligence Research* 48 (2013), 115–174.

[15] Andrea Calì, Domenico Lembo, and Riccardo Rosati. 2003. Query rewriting and answering under constraints in data integration systems. In *IJCAI*.

[16] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. 2007. Tractable Reasoning and Efficient Query Answering in Description Logics: The *DL-Lite* Family. *J. Autom. Reason.* 39, 3 (2007), 385–429.

[17] Hans de Nivelle. 1998. A Resolution Decision Procedure for the Guarded Fragment. In *CADE*.

[18] A. Deutsch, L. Popa, and V. Tannen. 2006. Query reformulation with constraints. *SIGMOD Record* 35, 1 (2006), 65–73.

[19] Harald Ganzinger and Hans De Nivelle. 1999. A Superposition Decision Procedure For the Guarded Fragment with Equality. In *LICS*.

[20] Georg Gottlob, Sebastian Rudolph, and Mantas Simkus. 2014. Expressiveness of guarded existential rule languages. In *PODS*.

[21] Alon Halevy, Anand Rajaraman, and Joann Ordille. 2006. Data Integration: The Teenage Years. In *VLDB*.

[22] Alon Y. Halevy. 2001. Answering queries using views: A survey. *VLDB Journal* 10, 4 (2001), 270–294.

[23] Colin Hirsch. 2002. Guarded Logics: Algorithms and Bisimulation. Available at http://www.umbrialogic.com/hirsch-thesis.pdf.

[24] Ullrich Hustadt, Boris Motik, and Ulrike Sattler. 2004. Reducing SHIQ-Description Logic to Disjunctive Datalog Programs. *KR* (2004).

[25] Ullrich Hustadt, Boris Motik, and Ulrike Sattler. 2007. Reasoning in description logics by a reduction to disjunctive datalog. *Journal of Automated Reasoning* 39, 3 (2007), 351–384.

[26] David S. Johnson and Anthony C. Klug. 1984. Testing Containment of Conjunctive Queries under Functional and Inclusion Dependencies. *JCSS* 28, 1 (1984), 167–189.

[27] Kevin Kappelmann. 2019. Decision Procedures for Guarded Logics. https://arxiv.org/abs/1911.03679.

[28] Alon Y. Levy. 2000. *Logic-Based Techniques in Data Integration*. 575–595.

[29] Thomas Lukasiewicz, Andrea Calì, and Georg Gottlob. 2012. A General Datalog-Based Framework for Tractable Query Answering over Ontologies. *Journal of Web Semantics* 14, 0 (2012), 57–83.

[30] Bruno Marnette. 2012. Resolution and Datalog Rewriting Under Value Invention and Equality Constraints. *arXiv preprint arXiv:1212.0254* (2012). http://arxiv.org/abs/1212.0254.

[31] M. Meier. 2014. The backchase revisited. *VLDB J.* 23, 3 (2014), 495–516.

[32] Boris Motik. 2006. *Reasoning in description logics using resolution and deductive databases*. Ph.D. Dissertation. Karlsruhe Institute of Technology, Germany. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000003797

[33] Boris Motik. 2021. Kaon2. http://kaon2.semanticweb.org/.

[34] Oxford KR group. 2021. Oxford Ontology Library. http://krr-nas.cs.ox.ac.uk/ontologies/lib/.

[35] M. S. Paterson and M. N. Wegman. 1976. Linear Unification. In *STOC*.

[36] David A. Plaisted and Steven Greenbaum. 1986. A Structure-preserving Clause Form Translation. *Journal of Symbolic Computation* 2, 3 (1986), 293–304.

[37] J. A. Robinson. 1965. Automatic deduction with hyper-resolution. *International journal of computer mathematics* 1, 3 (1965), 227––234.

[38] John Alan Robinson. 1965. A machine-oriented logic based on the resolution principle. *JACM* 12, 1 (1965), 23–41.

[39] Stephan Schulz. 2013. Simple and Efficient Clause Subsumption with Feature Vector Indexing. In *Automated Reasoning and Mathematics*.

[40] Mark E. Stickel. 1989. *The Path-Indexing Method for Indexing Terms*. Technical Report. SRI.

[41] Moshe Y. Vardi. 1997. Why Is Modal Logic so Robustly Decidable?. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol. 31. American Mathematical Society, 149–184.

[42] Roberto De Virgilio, Giorgio Orsi, Letizia Tanca, and Riccardo Torlone. 2012. NYAYA: A System Supporting the Uniform Management of Large Sets of Semantic Data. In *ICDE*.

[43] Zhe Wang, Peng Xiao, Kewen Wang, Zhiqiang Zhuang, and Hai Wan. 2021. Query Answering for Existential Rules via Efficient Datalog Rewriting. In *IJCAI*.

PROOF OF THEOREM 4.2 : COMPLETENESS OF THE ONE-PASS CHASE

Recall the statement:

For every set of *GTGDs* in head normal form, for every tree-like chase sequence $T_0, \ldots, T_n$, there is a one-pass, one-fact chase sequence $\overline{T_0}(=T_0), \overline{T_1}, \ldots, \overline{T_m}$ such that there is a homomorphism $h : T_n \longrightarrow \overline{T_m}$ with $h(c) = c$ for any constant c in T_0.

In particular, a *BCQ* Q has a chase proof from I and GTGDs Σ if and only if it has a one-pass, one-fact chase proof from I and Σ.

Proof. The argument here is a variant of prior proofs in [4, 27], and the version here was done jointly with Antoine Amarilli.

Recall that one of the requirements for being one-pass is that the chase is *rootward-only*: there are no parent-to-child ("leafward") propagation steps. We will start by ensuring this property, which is the main difficulty.

We prove the result by induction on n, calling h_n the homomorphism produced for n. We will ensure inductively that our homomorphism h_n preserves the tree structure of T_n. That is, there is additionally an injective homomorphism h_n^{T} from the underlying tree of T_n to the underlying tree of $\overline{T_m}$ such that for each node v of T_n and each fact $G \in \mathsf{FactsOf}_n(v)$, the node $h_n^{\mathrm{T}}(v)$ contains the image fact of G obtained by mapping the elements of the fact following h_n.

For the base case $n = 0$, we simply set $\overline{T_0} := T_0$, take $m = n = 0$, and let both h_n and h_n^{T} be the identity.

For the inductive case, there are two possibilities. The first is that we performed a chase step when going from T_{n-1} to T_n. This inductive case is easy, since it does not involve propagation, which is the only thing we care about in this part of the process. Assume we fire a trigger τ at a node v to create facts F, then we simply take the image $h_{n-1}(\tau)$ by the homomorphism in the node $h_{n-1}^{\mathrm{T}}(v)$ and fire it there, creating the facts $h_{n-1}(F)$ that we can use to extend the homomorphism.

The second possibility is the interesting one. We have performed a propagation step to go from T_{n-1} to T_n, and we must explain how to mimic it in $\overline{T_m}$ while only performing rootward propagation. To simplify the argument, letting F be one of the facts that were just propagated from a node v in T_{n-1} to nodes in T_n, we assume that the propagation from T_{n-1} to T_n propagates F to all nodes that have a guard for its elements. This is the most challenging case.

We first perform the rootward component of the propagation in $\overline{T_m}$. That is, we consider the ancestors of v having a guard for the elements of F, take their image by h_{n-1}^{T}, and propagate $\overline{F} = h_{n-1}(F)$ to these ancestors. Let $\overline{p}$ be the highest such ancestor in $\overline{T_m}$.

Now we simulate the leafwards component of the propagation by simply re-creating all the descendants of $\overline{p}$, which will automatically propagate the fact $\overline{F}$ to them. More precisely, consider U the set of all domain elements that occur in strict descendants of $\overline{p}$ but do not occur in $\overline{p}$, fix U' a disjoint set of fresh element names of the same cardinality, and fix a bijection h which maps U to U' and is the identity on the other elements of $\overline{T_m}$. Now, consider the sequence $\overline{T_0}, \ldots, \overline{T_m}$ obtained thus far, which by induction does not contain any downwards propagation. Now, re-play that sequence but replacing the elements of U by U'. More formally, all triggers and all created facts are mapped through h. In particular, the non-full chase steps that created the children of $\overline{p}$ will now create fresh child nodes, where the elements of U have been replaced by elements of U', further non-full chase steps on these children will continue creating a copy of their subtree, and Datalog chase steps happening in their subtrees as well as upwards propagations are performed in the same way as in the original sequence.

After this process, we have extended $\overline{T_0}, \ldots, \overline{T_m}$ by a sequence $\overline{T_{m+1}}, \ldots, \overline{T_{m'}}$, and $\overline{T_{m'}}$ is a superinstance of $\overline{T_m}$ which differs in that every child of $\overline{p}$ now exists in two copies, one featuring elements of U and the other one featuring the corresponding elements of U', these two copies being the roots of subtrees between which h is an isomorphism. Overall, the homomorphism h maps $\overline{T_m}$ to $\overline{T_{m'}}$ by mapping each original subtree to its copy. Let us call h^{T} the corresponding injective homomorphism at the level of tree nodes. We can compose h_m and h_m^{T} with h and h^{T} respectively to obtain well-defined homomorphisms $h_{m'}$ and $h_{m'}^{\mathrm{T}}$. Now, to show that they are suitable, the only point to verify is that we have correctly propagated the new fact. That is, for all nodes v' of T_n where F is guarded, the node $h_{m'}^{\mathrm{T}}$ indeed contains $\overline{F}$.

To understand why, notice that when we perform the sequence $\overline{T_{m+1}}, \ldots, \overline{T_{m'}}$, the node $\overline{p}$ contains the new fact $\overline{F}$ that we wished to downwards propagate. Hence, while we created the new subtrees, $\overline{F}$ was added to every new child node whenever it was guarded by the elements of that node. Thus, $\overline{F}$ now exists in each node of the new subtrees where it is guarded. This establishes that the chase sequence $\overline{T_0}, \ldots, \overline{T_{m'}}$ and the homomorphisms $h_{m'}$ and $h_{m'}^{\mathrm{T}}$ satisfy the conditions. This process can be iterated if facts other than F were propagated from T_{i-1} to T_n. This shows the inductive case, and concludes the inductive proof, establishing the result.

With this rootward-only normalisation in mind, we turn to achieving the one-pass property. This will not require any replaying, just simple re-ordering of steps. We will assume that in our rootward-only chase, we propagate rootward only from parent to child. This can be achieved by slowing down a step propagating to a descendant to multiple steps from parent to child. In the process we will be adding additional facts in intermediate nodes, but this does not break the rootward-only property. We prove the result by induction on n, calling h_n the isomorphism produced for n. We will ensure inductively that our homomorphism h_n preserves the tree structure of T_n. That is, let $\overline{T_0} \ldots, \overline{T_m}$ be the corresponding evolving one-pass chase proof, there is an isomorphism h_n^{T} from T_n to $\overline{T_m}$ such that for each fact $G \in T_n$ associated with node a in T_n, the node $h_n^T(a)$ contains the image fact of G obtained by mapping the elements of the fact following h_n.

For the base case $n = 0$, we simply set $\overline{T_0} := T_0$ letting h_n and the corresponding tree mapping h_n^T both be the identity.

Now suppose we have the result up to $n - 1$, and consider a a sequence of chase trees $T_0 \ldots T_n$. We apply the inductive hypothesis to $T_0 \ldots T_{n-1}$ to obtain: a one-pass chase sequence $\overline{T_0} \ldots, \overline{T_m}$, a homomorphism h_{n-1}, and a tree isomorphism h_{n-1}^T. For brevity, for any node v in T_m, we write $\overline{v}$ to denote $h_{n-1}^{\mathrm{T}}(v)$

There are now two cases. The first case is when the transition from T_{n-1} to T_n corresponds to applying a chase step, firings a trigger h for some dependency τ_n in some vertex v of the tree T_{n-1}. Let $\bar{v}$ be the corresponding node in $\overline{T_m}$. If $\bar{v}$ is an ancestor or equal to the recently-updated node in $\overline{T_m}$ we can just perform the same chase step to complete the induction. Otherwise let $\overline{T_i}$ be the last entry where $\bar{v}$ was updated prior to $\overline{T_m}$. By definition, all the facts in the image of h were already present in $\bar{v}$ within $\overline{T_i}$. We modify $\overline{T_0} \dots, \overline{T_m}$ by adding the same chase step after $\overline{T_i}$ as the new $(i+1)^{th}$ entry, call it T'_{i+1}. We thus add one to all the subsequent indices. If τ_n is a Datalog TGD, this will change the facts propagated by chase steps that come afterwards, but we simply modify those subsequent trees to carry along the additional facts. Since the recently-updated node of T'_{i+1} is either $\bar{v}$ or a child of $\bar{v}$, the resulting sequence is still one-pass.

The other case is when the transition from T_{n-1} to T_n involves propagation of a set of facts F from a node c to its parent p. We let $\bar{c}$ and $\bar{p}$ be the images of these nodes in the resulting chase. Again if $\bar{c}$ is already an ancestor or equal to the recently-updated node of $\overline{T_m}$, we simply append the corresponding propagation step. Otherwise we find the last place where $\bar{c}$ was updated prior to this point in the sequence, and we add the same propagation step directly afterwards. As in the case above, since we are adding facts on to $\bar{p}$, we may need to adjust the chase trees in subsequent steps accordingly to account for this.

The modification to achieve the one-fact property is straightforward. We can assume we have already achieved the one-pass property and then inductively adjust to get one-fact. Suppose in some inductive step we arrive at a final step from T_n to T_{n+1} which propagates multiple facts $F_1 \dots F_k$ from node c to p. Then we propagate only one of them, say F_1, then regenerate a copy of c as above, and propagate F_2 from the copy to p. We iterate until we have propagated all of the facts to p.

□

PROOF OF PROPOSITION 4.3: CHARACTERIZING REWRITING BASED ON THE ONE-PASS CHASE

Recall the statement:

Consider a set Σ' of Datalog rules that are logical consequences of Σ. Suppose that for each simple loop $T_0, \dots, T_n$ of Σ with output fact F, there is a TGD in Σ' which when applied to T_0 produces F. Then Σ' is a rewriting of Σ.

Note that the condition is also clearly necessary for Σ' to be a rewriting since a rewriting must produce all ground consequences of T_0.

Proof. It suffices to show that if a ground fact A is produced in the chase of I with Σ, then it is also produced by applying Σ' to I. By completeness of the one-pass one-fact chase, we can assume that A is produced by a one-pass and one-fact chase $T_0, \dots, T_n$. We track the evolution of the root in the chase, letting $T_{j_1}, \dots, T_{j_k}$ be the indices where the root is modified, and let $j_0 = 0$. Then for each i the sequences between T_{j_i} and $T_{j_{i+1}}$ form a simple loop, and by hypothesis, there are firings of rules in Σ' that can realize these evolutions. Thus composing these firings gives us the chase with Σ' that we need. □

LEMMAS CONCERNING UNIFICATION

In the remaining appendix proofs, we will use some well-known facts about unification.

Lemma .1. *For any unifiable atoms $A_1, \dots, A_n$ and $B_1, \dots, B_n$, the mgu is unique up to renaming.*

Proof. Let θ, θ' be two mgus of the given atoms. Then there are substitutions δ, δ' such that $\theta = \theta'\delta'$ and $\theta' = \theta\delta$. Thus, $\theta = \theta\delta\delta'$ and $\theta' = \theta'\delta'\delta$, i.e. $\delta\delta' = id$ and $\delta'\delta = id$. Hence, δ, δ' are mutual inverse bijections, and thus renamings. □

Many different unification algorithms, some even linear, exist.

Lemma .2. *The mgu of atoms $A_1, \dots, A_n$ and $B_1, \dots, B_n$ can be obtained in time $O\big(\sum_{i=1}^{n} |A_i| + |B_i|\big)$, where $|A|$ is the encoding size of an atom A.*

Proof. A linear procedure can be found, for example, in [35]. □

Lemma .3. *Assume $A_1, \dots, A_n$ and $B_1, \dots, B_n$ are unifiable and function-free, and let θ be an mgu. Then all $\theta(A_i) = \theta(B_i)$ are function-free.*

Proof. Assume otherwise. Pick some variable x. Since all given atoms are function-free and unifiable, the substitution θ' defined by $\theta'(v) := x$ for any $v \in \mathcal{V}$ is a unifier of given atoms and all $\theta'(A_i) = \theta'(B_i)$ are function-free. Thus, there is a substitution δ such that $\theta' = \theta\delta$. Pick some A_i such that $\theta(A_i)$ contains a functional subterm $f(\vec{t})$. Then $\delta(f(\vec{t})) = f(\delta(\vec{t}))$ and hence, $\theta'(A_i) = \theta\delta(A_i)$ is functional, a contradiction. □

For our Skolemized algorithms, we need a stronger statement:

Lemma .4. *Assume $A_1, \dots, A_n$ and $B_1, \dots, B_n$ are shallow (i.e. contain either no functions or only functions at depth 1) and each function contains all variables of the atom. Then applying the mgu to each atom again gives a shallow atom.*

Proof. cf. Theorem 2 in [17]. □

PROPERTIES OF SimDR

We will fill in the details of some claims concerning SimDR in the paper. Although SimDR is not a competitive rewriting algorithm, many of the techniques here will be relevant to the other correctness claims.

Some basic properties of SimDR

Lemma .5. *Assume we perform an* (ORIGINAL) *inference on τ, τ' with θ the unifier. Then $\theta(\mathsf{vars}(B')) \cap \vec{y} \neq \emptyset$ for any B' in S'.*

Proof. Recall that (ORIGINAL) requires that θ is the identity on $\vec{y}$. Thus any atom in $\theta(\eta)$ contains some y_i. As any B' in S' unifies with some atom in η using θ, this implies $\mathsf{vars}(\theta(B')) \cap \vec{y} \neq \emptyset$. ◻

Corollary .6. *Assume we perform an* (ORIGINAL) *inference on τ, τ' with θ the unifier. Then B' occurs in S' if and only if $\mathsf{vars}(\theta(B')) \cap \vec{y} \neq \emptyset$.*

Proof. Left to right is Lemma .5. Right to left follows from the variable check that is included in (ORIGINAL). ◻

Correctness of SimDR

We sketch how correctness of SimDR is proven, by using the invariant of Proposition 4.3. The full proof follows the structure of the corresponding argument for ExbDR, given in the next section.

Consider a simple loop in a one-pass one-fact chase sequence of Σ with trees $T_0, \ldots, T_n$, where T_0 consists of a single root node r and T_n adds on a single fact F to r. By the proposition, it suffices to simulate the impact of this loop with a Datalog rule produced by the rewriting. We work by induction on the length of the sequence.

One possibility is that $T_n = T_1$ is the result of applying a Datalog rule on T_0. In this case the result is immediate.

Thus we can assume T_1 is obtained from T_0 by applying a non-full TGD $\tau \in \Sigma$, adding on a child node c to r. Let $T_1', \ldots, T_n'$ be the sequence $T_1, \ldots, T_n$ with r removed from each tree. This is not necessarily a simple loop, but can be decomposed into several simple loops. By induction, each such simple loop is given by a Datalog rule in $\mathsf{SimDR}(\Sigma)$. Now by repeated applications of (COMPOSE) we can get a single dependency ρ in $\mathsf{SimDR}(\Sigma)$ that generates F from T_1. Now we use (ORIGINAL) to compose τ and ρ to get a rule that creates F from T_0.

Complexity of SimDR

We give the complexity analysis of SimDR in some detail, showing that it meets worst-case bounds given for query answering [14]. A similar analysis applies to our other algorithms.

In the following, let n be the number of predicate symbols in Σ, a be the maximum arity of any predicate, let $c := |\mathsf{consts}(\Sigma)|$, and let $w := \mathsf{width}(\Sigma) + c$.

We first note that all TGDs in SimDR are of a bounded width:

Lemma .7. *Any $\sigma \in \mathsf{SimDR}(\Sigma)$ is a Datalog rule with $\mathsf{width}(\sigma) \leq w$.*

Proof. We show the claim by induction. Clearly, all rules in Σ satisfy the conditions.

If we perform a (COMPOSE) step to derive σ, then σ will only contain variables among $\{x_1, \ldots, x_w\}$, and hence $\mathsf{width}(\sigma) \leq w$. Since the rules used to derive σ are Datalog rules, σ will also be a Datalog rule.

If we perform an (ORIGINAL) step to derive $\sigma = \beta'' \rightarrow \eta''$, the used rules τ, τ' are of the form

$$\beta(\vec{x}) \rightarrow \exists \vec{y}\, \eta(\vec{x}, \vec{y}),$$
$$\beta'(\vec{z}) \rightarrow \eta'(\vec{z}).$$

By definition, $\theta(\eta'')$ contains no y_i. Thus σ will be a Datalog rule. Further, by the conditions imposed on (ORIGINAL), we have $\mathsf{vars}(\theta(\beta')) \subseteq \theta(\vec{x}) \cup \vec{y}$. By Corollary .6, this implies $\mathsf{vars}(\theta(\beta' \setminus \{B_1', \ldots, B_n'\})) \subseteq \theta(\vec{x})$. Hence $\mathsf{vars}(\beta'') \subseteq \theta(\vec{x})$. As $\tau \in \Sigma$, we have $|\theta(\vec{x})| \leq w$, and hence $\mathsf{width}(\sigma) \leq w$. ◻

Lemma .8. *Any TGD τ contains at most $n\big((w+c)^a + (w+c)^a\big)$ atoms.*

Proof. For any fixed predicate symbol, we have at most $\big(\mathsf{bwidth}(\tau) + c\big)^a$ and $\big(\mathsf{hwidth}(\tau) + c\big)^a$ different instantiations in the body and head of τ, respectively. ◻

Lemma .9. $|\mathsf{SimDR}(\Sigma)| \leq 2^{2n(w+c)^a}$.

Proof. Recall that SimDR normalizes TGDs in its output so that variables have a canonical ordering. Now from above bounds, we infer that there are at most $n((w+c)^a + (w+c)^a) = 2n(w+c)^a$ different atoms in $\mathsf{SimDR}(\Sigma)$. Each TGD in $\mathsf{SimDR}(\Sigma)$ consists of a subset of these atoms, which gives us the claimed bound. ◻

Lemma .10. *The* SimDR *algorithm terminates in* PTIME *for bounded w, n, a, c,* EXPTIME *for bounded a, and* 2EXPTIME *otherwise.*

Proof. The algorithm takes all pairs in $\Sigma \times \mathsf{SimDR}(\Sigma)$ and $\mathsf{SimDR}(\Sigma)^2$ to derive all possible resolvents of any pair. By Lemma .9, this process takes at most $s := 2^{2n(w+c)^a}$ iterations, and at any point, there are at most $O(s^2 + |\Sigma|s)$ pairs.

For the sake of simplicity, let us be pessimistic and assume that (ORIGINAL) enumerates all possible unifiers rather than just considering mgus. There are at most $(w+c)^{2w}$ potential unifiers when considering two rules. At any step, we have to apply the unifier and find the matching head and body atom pairs of the two considered rules. By Lemmas .7 and .8, there are at most $\left(n(w+c)^a\right)^{n(w+c)^a}$ such pairs. If some atoms unify (and the additional conditions of (ORIGINAL) are satisfied), we resolve the rules and normalise the result to obtain a new rule. The variable and head normal form transformations can trivially be done in linear time and linear-time unification exists by Lemma .2. Thus, we obtain the claimed complexity bounds. □

PROPERTIES OF ExbDR

Correctness of ExbDR

Our goal is to prove Theorem 5.8, the correctness of ExbDR:

ExbDR(Σ) is a rewriting of Σ.

We will focus first on the naïve version which just performs the (G-RESOLVE) rule until a fixed point is reached, and discuss the algorithm in detail afterwards.

We again verify the property of Proposition 4.3, fixing a one-pass one-fact chase sequence $T_0, \dots, T_n$ that forms a simple loop adding fact F to T_0. In the base case for the invariant, the simple loop is the firing of a full TGD of Σ on T_0, and the result is again clear.

In the inductive case for the invariant, F was added as a result of firing a TGD on a descendant d of the root, and then was propagated to the root. In this case T_1 must have been formed by applying a non-full TGD τ on the root, creating a child c. Let $T'_1, \dots, T'_n$ be the trees T_i with the root removed, thus trees rooted at c. The evolution of c in this sequence breaks down into some number k of simple loops sl'_i, with the last loop resulting in the adding of F to c. The situation is shown Figure 4 in the body of the paper.

We can apply the induction hypothesis to each of these simple loops, simulating each of them by a full TGD σ'_i in ExbDR.

Informally, we will apply (G-RESOLVE) with σ'_1 and non-full rule τ to get a non-full TGD τ'_1 in the intermediate rewriting that produces the version of c after the first loop. Similarly, we can continue applying (G-RESOLVE) with τ'_i and δ'_{i+1} until finally we obtain τ'_k that produces the final version of node c in the loop which will produce F starting at c.

Let us formalize the first step. We write τ and σ'_1 as:

$$\begin{aligned}\tau &= \beta(\vec{x}) \to \exists \vec{y}\, \eta(\vec{x}, \vec{y}),\\ \sigma'_1 &= \beta'(\vec{z}) \to \eta'(\vec{z}),\end{aligned}$$

and assume the dependencies are in head normal form. Let h_σ be the trigger used in applying σ'_1 to realize the first loop, and let h_τ be the trigger on τ used to create the child c. Note that if $h_\sigma(\beta') \subseteq T_0$ – the first loop only used facts inherited from from the parent – then the output of the first loop must be a new fact over T_0, which can only be F. So we could use σ'_1 directly for our final dependency. So assume the chase step with σ'_1 makes use of some facts generated by τ in the child c. Let $B'_1, \dots, B'_n$ be all atoms in β' such that for each $1 \le i \le n$, there is $H_i \in \eta$ with $h_\tau(H_i) = h_\delta(B'_i)$. Then the substitution θ defined by

$$\theta(v) := \begin{cases} x_i, & \text{if } v \in \vec{x} \text{ and } h_\tau(v) = c_i \\ x_i, & \text{if } v \in \vec{z} \text{ and } h_\delta(v) = c_i \\ y_i, & \text{if } v \in \vec{z} \text{ and } h_\delta(v) = h_\tau(y_i) \end{cases}$$

is a unifier of $H_1, \dots, H_n$ and $B'_1, \dots, B'_n$ that is the identity on $\vec{y}$ and $\theta(\vec{x}) \cap \vec{y} = \emptyset$. Hence, there is an mgu θ^* of $H_1, \dots, H_n$ and $B'_1, \dots, B'_n$ that is the identity on $\vec{y}$ (by treating $\vec{y}$ as constants). Thus, there is a substitution δ such that $\theta^* = \theta\delta$. By Lemma .21, δ is the identity on $\vec{y}$.

Claim .11. $\theta^*(\vec{x}) \cap \vec{y} = \emptyset$.

Proof. Assume otherwise. Then for some $x_i \in \vec{x}$, we get $\theta^*(x_i) = y_j$, and thus $\theta(x_i) = \theta^*\delta(x_i) = \delta(y_j) \notin \vec{y}$ since $\theta(\vec{x}) \cap \vec{y} = \emptyset$, contradicting the fact that δ is the identity on $\vec{y}$. □

Claim .12. *Any* $B' \in \beta'$ *with* $\mathsf{vars}(\theta^*(B')) \cap \vec{y} \neq \emptyset$ *occurs in* $B'_1, \dots, B'_n$.

Proof. If $\mathsf{vars}(\theta^*(B')) \cap \vec{y} \neq \emptyset$, then also $\mathsf{vars}(\theta(B')) = \delta(\mathsf{vars}(\theta^*(B'))) \cap \vec{y} \neq \emptyset$ as δ is the identity on $\vec{y}$. Thus, by construction of θ, $\mathsf{consts}(h_\delta(B')) \cap h_\tau(\vec{y}) \neq \emptyset$, and hence, $h_\delta(B') \in T_1 \setminus T_0 \subseteq h(\eta)$. □

Using Claim .11 and .12, we can derive that θ^* satisfies the existential variables check. Now define

$$\begin{aligned}\beta'' &:= \theta\big(\beta \cup (\beta' \setminus \{B'_1, \dots, B'_n\})\big),\\ \eta'' &:= \theta(\eta \cup \eta').\end{aligned}$$

Then using (G-RESOLVE), we obtain $\tau'_1 := \beta'' \to \eta''$. Let h' be the trigger defined by $h'(x_i) := c_i$ and $h'(y_i) = h_\tau(y_i)$ and set $h^* := \delta h'$.

Claim .13. *For any $v \in \vec{x} \cup \vec{y}$, we have $h'(\theta(v)) = h_\tau(v)$, and for any $z_i \in \vec{z}$, we have $h'(\theta(z_i)) = h_\sigma(z_i)$.*

Proof. Pick some $x_i \in \vec{x}$ and assume $h_\tau(x_i) = c_j$. Then $\theta(x_i) = x_j$ by definition of θ. Further, by definition of h', $h'(x_j) = c_j$. Hence, $h'(\theta(x_i)) = h'(x_j) = c_j = h_\tau(x_i)$.

Now pick some $y_i \in \vec{y}$. Then $\theta(y_i) = y_i$, and by definition of h', $h'(y_i) = h_\tau(y_i)$. Hence, $h'(\theta(y_i)) = h'(y_i) = h_\tau(y_i)$.

Lastly, pick some $z_i \in \vec{z}$. The case $h_\sigma(z_i) = c_j$ is analogous to the first case. Finally consider the case $h_\sigma(z_i) = h_\tau(y_k)$. Then $\theta(z_i) = y_k$ by definition of θ. Hence, $h'(\theta(z_i)) = h'(y_k) = h_\tau(y_k) = h_\sigma(z_i)$. □

Claim .14. *$h^*(\beta'') \subseteq T_0$.*

Proof. We have

$$\begin{aligned} h^*(\beta'') &= \delta h'(\beta'') = h'\Big(\theta^*\delta\big(\beta \cup (\beta' \setminus \{B'_1, \dots, B'_n\})\big)\Big) \\ &= h'\Big(\theta\big(\beta \cup (\beta' \setminus \{B'_1, \dots, B'_n\})\big)\Big) = h'\Big(\theta(\beta) \cup \theta\big(\beta' \setminus \{B'_1, \dots, B'_n\}\big)\Big) \\ &= h'(\theta(\beta)) \cup h'\big(\theta(\beta' \setminus \{B'_1, \dots, B'_n\})\big) \overset{Claim .13}{=} h_\tau(\beta) \cup h_\delta\big(\beta' \setminus \{B'_1, \dots, B'_n\}\big) \\ &\subseteq T_0 \cup (T_1 \setminus h_\tau(\eta)) = T_0, \end{aligned}$$

where the last step follows from the choice of $B'_1, \dots, B'_n$; that is, if $h_\delta(B') \in T_1 \setminus T_0 \subseteq h_\tau(\eta)$ for some $B' \in \beta'$, then B' occurs in $B'_1, \dots, B'_n$. □

Claim .15. *$h^*(\eta'') = h_\tau(\eta) \cup h_\delta(\eta')$.*

Proof. We have

$$h^*(\eta'') = \delta h'(\eta'') = h'\big(\theta^*\delta(\eta \cup \eta')\big) = h'\big(\theta(\eta) \cup \theta(\eta')\big) = h'\big(\theta(\eta)\big) \cup h'\big(\theta(\eta')\big) \overset{Claim .13}{=} h_\tau(\eta) \cup h_\delta(\eta').$$

□

Using Claims .14 and .15, we derive that firing τ'' based on h^* in T_0 creates $h_\tau(\eta) \cup h_\delta(\eta')$.

Similarly, we can continue applying (G-RESOLVE) with τ'_i and σ'_{i+1} until finally we obtain τ'_k that produces the final version of node c including F when fired in T_0. Breaking this τ'_k down into head normal form, we will get a full TGD that produces F from T_0 as required.

Proof of Proposition 5.9: Selecting the Sequences S and S'

To obtain a practical implementation of ExbDR, one needs to decide how to obtain the sequences S, S' used in (G-RESOLVE). The aim of this section is to prove Proposition 5.9, which shows us that we can effectively retrieve and limit the number of potentially matching sequences. Recall the statement:

Assume we perform a (G-RESOLVE) inference on τ, τ' and θ^* is the mgu of a head atom of τ with the guard atom of τ' that is the identity on $\vec{y}$. Then B' occurs in S' if and only if $\mathsf{vars}(\theta^*(B')) \cap \vec{y} \neq \emptyset$.

We start with some simple observations about (G-RESOLVE).

Lemma .16. (G-RESOLVE) *inferences are permutation independent of S, S'. More precisely, for any $S = H_1, \dots, H_n$, $S' = B'_1, \dots, B'_n$, and permutation $\sigma \in S_n$, where S_n is the symmetric group with n elements, the rules derived using* (G-RESOLVE) *under S, S' and $H_{\sigma(1)}, \dots, H_{\sigma(n)}$, $B'_{\sigma(1)}, \dots, B'_{\sigma(n)}$ coincide.*

Proof. The existential variable check (evc) does not consider the order of the sequences and the mgus of permuted sequences coincide. □

The next lemma and corollary are proven just as their analogs for (ORIGINAL) in the prior section.

Lemma .17. *Assume we perform a* (G-RESOLVE) *inference on $\tau, \tau' \in \mathsf{IExbDR}(\Sigma)$ with θ the unifier. Then $\theta(\mathsf{vars}(B')) \cap \vec{y} \neq \emptyset$ for any B' in S'.*

Corollary .18. *Assume we perform a* (G-RESOLVE) *inference on $\tau, \tau' \in \mathsf{IExbDR}(\Sigma)$ with θ the unifier. Then B' occurs in S' if and only if $\mathsf{vars}(\theta(B')) \cap \vec{y} \neq \emptyset$.*

The next lemma formalizes the intuition that the guard of τ' needs to be involved in a (G-RESOLVE) step.

Lemma .19. *Assume we perform a* (G-RESOLVE) *inference on τ and τ' and τ' is guarded by G'. Then G' occurs in S'.*

Proof. Let B' be an atom in S' that unifies with some H in S as part of the inference step with unifier θ. Then $\theta(H)$ must contain some y_i since θ is the identity on $\vec{y}$. As B' unifies with H, we get $\mathsf{vars}(\theta(B')) \cap \vec{y} \neq \emptyset$. Finally, since G' guards τ', we get $\theta(\mathsf{vars}(B') \subseteq \mathsf{vars}(\theta(G'))$ and hence $\mathsf{vars}(\theta(G')) \cap \vec{y} \neq \emptyset$. Thus, by the evc, G' occurs in S'. □

Thus we now know that the guard G' of τ' must occur in S', and thus in particular it unifies with some atom in the head of τ. The corollary above tells us what the set of atoms S' in τ' must be, assuming we knew what the unifier was. The next proposition, which corresponds to Proposition 5.9, tells us that we need only consider the mgu of a head atom of τ with a guard of τ'.

PROPOSITION .20. *Assume we perform a* (G-RESOLVE) *inference on* $\tau = \beta(\vec{x}) \longrightarrow \exists \vec{y}\, \eta$ *and* $\tau' = \beta' \longrightarrow \eta'$ *and* θ^* *is the mgu of a head atom of* τ *with the guard atom of* τ' *that is the identity on* $\vec{y}$*. Then* B' *occurs in* S' *if and only if* $\mathrm{vars}(\theta^*(B')) \cap \vec{y} \neq \emptyset$.

PROOF. Assume some $S = H, H_2, \ldots, H_n$, $S' = G', B'_2, \ldots, B'_n$, and θ satisfy all restrictions necessary for an inference. In particular, $\theta(H) = \theta(G')$ and θ is the identity on $\vec{y}$. Thus there must be δ such that $\theta = \theta^*\delta$.

CLAIM .21. *δ is the identity on $\vec{y}$*

SUBPROOF. Assume otherwise. Then there is $y_i \in \vec{y}$ with $\delta(y_i) \neq y_i$. Now as θ, θ^* are the identity on $\vec{y}$, we get $y_i = \theta(y_i) = \theta^*\delta(y_i) = \delta(y_i) \neq y_i$, a contradiction. ■

CLAIM .22. *For any $v \in \mathrm{vars}(\beta')$, we have $\theta(v) \in \vec{y}$ if and only if $\theta^*(v) \in \vec{y}$.*

SUBPROOF. Since G' guards τ', we have $\theta^*(v) \in \mathrm{vars}(\theta^*(G')) = \mathrm{vars}(\theta^*(H)) \subseteq \theta^*(\vec{x}) \cup \vec{y}$. Now if $\theta^*(v) \in \vec{y}$, then also $\theta^*\delta(v) = \theta(v) \in \vec{y}$ as δ is the identity on $\vec{y}$. If $\theta^*(v) \in \theta^*(\vec{x})$, then $\theta^*\delta(v) \in \theta^*\delta(\vec{x}) = \theta(\vec{x})$, and as $\theta(\vec{x}) \cap \vec{y} = \emptyset$, this implies $\theta^*\delta(v) = \theta(v) \notin \vec{y}$. ■

Now by Corollary .18, B' occurs in S' if and only if $\mathrm{vars}(\theta(B')) \cap \vec{y} \neq \emptyset$, which by Claim .22 is the case if and only if $\mathrm{vars}(\theta^*(B')) \cap \vec{y} \neq \emptyset$. □

Thus we have only one possibility for S' once we have unified the guard. We discuss the choices for S in the next section.

Details of the ExbDR Algorithm

Based on the analysis of the possible combinations to consider in the previous section, we now provide the full ExbDR algorithm: see Algorithm 2.

Algorithm 2: More detailed look at ExbDR.

```
Function ExbDR:
    Input  : Guarded Datalog and non-full TGDs $\Sigma_f \cup \Sigma_n$
    Output : Rewriting of $\Sigma$
    $\mathcal{W} = \Sigma_n$                                   // working set
    $\mathcal{D} = \Sigma_f$                                   // Datalog set
    $\mathcal{N} = \emptyset$                                  // non-full set
    while $\mathcal{W} \neq \emptyset$ do
        Let $\sigma$ be a rule in $\mathcal{W}$
        if $\sigma$ is non-full then
            $\mathcal{N} = \mathcal{N} \cup \{\sigma\}$
            $\mathcal{E} = \{\text{(G-RESOLVE)}(\sigma, \tau) \mid \tau \in \mathcal{D}\}$
        else
            $\mathcal{D} = \mathcal{D} \cup \{\sigma\}$
            $\mathcal{E} = \{\text{(G-RESOLVE)}(\tau, \sigma) \mid \tau \in \mathcal{N}\}$
        end
        $\mathcal{W} = \mathcal{W} \cup \mathcal{E} \setminus (\mathcal{D} \cup \mathcal{N})$
    end
    return $\mathcal{D}$

Function (G-RESOLVE):
    : Non-full $\sigma = \beta(\vec{x}) \rightarrow \exists \vec{y}\, \eta$ and Datalog $\sigma' = \beta' \rightarrow \eta'$ Output : Derived rules of $\sigma$, $\sigma'$
    Rename $\sigma'$ s.t. $\text{vars}(\sigma) \cap \text{vars}(\sigma') = \emptyset$
    Let $G'$ be a guard of $\sigma'$
    $\mathcal{E} = \emptyset$
    foreach $H \in \eta$ do
        if there is an mgu $\theta$ of $G'$ and $H$ with $\theta|_{\vec{y}} = id_{\vec{y}}$ and $\theta(\vec{x}) \cap \vec{y} = \emptyset$ then
            Apply $\theta$ to $\sigma$
            Apply $\theta$ to $\sigma'$
            $S' = (G', B'_1, \ldots, B'_n)$ s.t. each $B' \in \beta'$ with $B' \cap \vec{y} \neq \emptyset$ occurs exactly once in $S'$
            for $1 \leq i \leq n$ do
                Let $B'_i = R(w_1, \ldots, w_m)$ and $S_i = \{R(v_1, \ldots, v_m) \in \eta \mid \forall 1 \leq j \leq m.\, v_j \in \vec{y} \vee w_j \in \vec{y} \implies v_j = w_j\}$
            end
            foreach $S \in (\{H\} \times S_1 \times \cdots \times S_n)$ do
                if there is an mgu $\theta^*$ of $S$ and $S'$ then
                    $\beta'' = \theta^*(\beta \cup (\beta' \setminus \{G', B'_1, \ldots, B'_n\}))$
                    $\eta'' = \theta^*(\eta \cup \eta')$
                    $\mathcal{E} = \mathcal{E} \cup (\beta'' \rightarrow \exists \vec{y}\, \eta'')$
                end
            end
        end
    end
    return $\mathcal{E}$
```

The top-level algorithm is the generic "saturation" loop – as mentioned in the body; we omit normalization and subsumption steps here.

The resolution algorithm chooses a guard atom for the righthand rule body and unifies it with the head of the left rule. If there is no unification then we can exit. This is justified by the results in the prior section.

Otherwise we fix the unifier and apply it to both rules. Thus the right hand rule's body now contains the variables on the head of the head: we will thus refer to "existentially quantified" variables (those that are existentially quantified in the head of the left rule) and "plain variables". We set S' to be all atoms in the righthand rule's body that contain an existentially quantified variable after the unification. Again, this is justified by the results in the previous section.

We now need to consider the options for S, and there can be many. We need to look at all "sound" ways of refining the unification to include each of these atoms: soundness means that we are allowed to unify variables on the right, as long as they are not existentially quantified. There may be many such unifications, and we consider "almost all" of them: we consider all of the possibilities for a given atom A_i' in S', and then for each choice of a possibility for each A_i' we take the most general unifier.

Complexity of ExbDR

Below, we will again silently assume that all TGDs are in head normal form unless stated otherwise. Let n be the number of predicate symbols in Σ, a be the maximum arity of any predicate, let $c := |\mathsf{consts}(\Sigma)|$, $w_b := \mathsf{bwidth}(\Sigma)$, $w_h := \mathsf{hwidth}(\Sigma)$, and $w := \max\{w_b, w_h\}$. As in the case for SimDR, we want to show that each rule in ExbDR is of bounded width.

Proposition .23. *Any $\sigma \in \mathsf{IExbDR}(\Sigma)$ is a GTGD with $\mathsf{bwidth}(\sigma) \leq w_b$, and $\mathsf{hwidth}(\sigma) \leq w_h$.*

Proof. We prove the claim inductively. Clearly, all rules in Σ satisfy the conditions.

Assume we create $\sigma = \beta'' \rightarrow \exists \vec{y}\, \eta''$ by (G-RESOLVE) from some non-full $\tau = \beta \rightarrow \exists \vec{y}\, \eta$ and Datalog $\delta' = \beta' \rightarrow \eta'$ in $\mathsf{IExbDR}(\Sigma)$. By the inductive hypothesis, β' is guarded by some G'. Hence, by Lemma .19, G' occurs in S'. Let H be the atom in S that is unified with G' using θ. By Lemma .3, θ does not introduce functions in σ. As G' guards β', $\theta(H) = \theta(G')$, and θ is the identity on $\vec{y}$, we get

$$\mathsf{vars}(\theta(\eta')) \subseteq \mathsf{vars}(\theta(\beta')) = \mathsf{vars}(\theta(G')) = \mathsf{vars}(\theta(H)) \subseteq \mathsf{vars}(\theta(\eta)) \subseteq \mathsf{vars}(\theta(\beta)) \cup \vec{y}.$$

From this, we can derive that $\mathsf{vars}(\beta'') \subseteq \mathsf{vars}(\theta(\beta)) \cup \vec{y}$ and $\mathsf{vars}(\eta'') = \mathsf{vars}(\theta(\eta))$. Since the evc is satisfied, the former refines to $\mathsf{vars}(\beta'') = \mathsf{vars}(\theta(\beta))$. By the inductive hypothesis, we have $\mathsf{bwidth}(\tau) \leq w_b$ and $\mathsf{hwidth}(\tau) \leq w_h$, and hence $\mathsf{bwidth}(\sigma) \leq w_b$ and $\mathsf{hwidth}(\sigma) \leq w_h$. Finally, again by the inductive hypothesis, τ is guarded by some G, which will also be a guard for σ. □

Corollary .24. $|\mathsf{IExbDR}(\Sigma)| \leq 2^{n\left((w_b+c)^a+(w_h+c)^a\right)}$.

Proof. Analogous to Lemma .9. □

Lemma .25. *Let $\tau, \tau' \in \mathsf{IExbDR}(\Sigma)$. Then all (G-RESOLVE)-resolvents of $\tau = \beta \rightarrow \eta$, $\tau' = \beta' \rightarrow \eta'$ can be obtained in PTIME when n, a, w, c are all bounded, EXPTIME for bounded a, and 2EXPTIME otherwise.*

Proof. Let $G' \in \beta'$ be a guard. Using our observations in Proposition .20, we first check for any $H \in \eta$ whether there is an mgu θ^* of G' and H that is the identity on $\vec{y}$ (by treating $\vec{y}$ as constants). We then select all atoms $B_2', \ldots, B_m' \in \beta' \setminus \{G'\}$ with $\mathsf{vars}(B_i'\theta^*) \cap \vec{y} \neq \emptyset$. These first steps can be done using a linear time unification procedure. Also note that the exact choice of G' and the order of $S' := G', B_2', \ldots, B_m'$ are irrelevant by Lemma .19 and Lemma .16.

Now for any $B_i' \neq G'$, we have at most $|\eta|$ possible counterparts in η. We check for any possible combination S if we can extend θ^* to an mgu θ of S, S' that is the identity on $\vec{y}$ and $\vec{x}\theta \cap \vec{y} = \emptyset$. Finally, we resolve the rules and normalise the result to obtain a new rule. Since we can normalise TGDs in linear time, we see that these steps can again be done in linear time. Since τ contains at most $n \cdot (w_b + c)^a$ body atoms and τ' at most $n \cdot (w_h + c)^a$ head atoms, we have to repeat above steps at most $(n \cdot (w_h + c)^a)^{(n \cdot (w_b+c)^a)}$ times. Hence, we obtain the claimed complexity bounds. □

Lemma .26. *The algorithm for computing ExbDR algorithm terminates in PTIME for bounded w, n, a, c, EXPTIME for bounded a, and 2EXPTIME otherwise.*

Proof. The algorithm takes all pairs in $\mathsf{IExbDR}(\Sigma)$ and performs all (G-RESOLVE)-inferences. By Corollary .24, this process takes at most $s := 2^{n\left((w_b+c)^a+(w_h+c)^a\right)}$ iterations, and at any point, there are at most s^2 pairs. Finally, by Lemma .25, the complexity for each τ, τ' falls in the stated bounds. □

PROPERTIES OF SkolemDR

Correctness of SkolemDR

We sketch the correctness proof of SkolemDR as we did for SimDR by using Proposition 4.3. The full proof follows the structure of the corresponding argument for ExbDR, which was illustrated in Figure 4 in the body of the paper.

Consider a simple loop in a one-pass one-fact chase sequence of Σ with trees $T_0, \ldots, T_n$, where T_0 consists of a single root node r and T_n adds on a single fact F to r. We work by induction on the length of the sequence. The base case, where $n = 0$, again follows trivially, so let us focus on the inductive case.

We can easily reduce to the case where T_1 is obtained from T_0 by applying a non-full TGD $\tau \in \Sigma$, adding on a child node c to r. Let $T_1', \ldots, T_n'$ be the sequence $T_1, \ldots, T_n$ with r removed from each tree. Again, this is not necessarily a simple loop, but can be decomposed into m simple loops. By induction, each such simple loop s_l^i is given by a Datalog rule $\sigma_i \in \mathsf{SkolemDR}(\Sigma)$. Moreover, the Skolemisation of the non-full TGD τ gives us a set of body-full but non-full Skolemized TGDs $\tau_1, \dots, \tau_k \in \mathsf{SkolemDR}(\Sigma)$. Now the firing of σ_1 uses a subset $F_1, \dots, F_l$ of all facts produced by firing τ. Each such fact F_i corresponds to a head atom H_i of the non-full TGD τ, which in turn corresponds to the head of one of the Skolemized TGDs τ_{i_i}. Now iteratively applying (SK-RESOLVE) to σ_1 and all such $\tau_{i_1}, \dots, \tau_{i_l}$ gives us a body-full τ_{k+1} that creates the (Skolemized) fact of the simple loop s_l^1 when fired in T_0. Similarly, applying (SK-RESOLVE) to σ_2 and a subset of $\tau_1, \dots, \tau_{k+1}$ gives a body-full τ_{k+2} that creates s_l^2, and so on, until we finally obtain the Datalog τ_{k+m} that creates the fact of s_l^m when fired in T_0.

SIMULATING SkolemDR **VIA** ExbDR

In the body of the paper we provided examples where the number of TGDs produced by ExbDR is exponentially larger than the number in SkolemDR. See Example 5.10 and Example 5.12. Thus SkolemDR can improve exponentially over ExbDR. An obvious question is whether there are examples where the opposite happens: SkolemDR produces exponentially more rules than ExbDR. We show that this is in fact impossible: the size of SkolemDR is always bounded by a polynomial in the output size of ExbDR. In fact, any derivation we get in SkolemDR can be simulated by one in ExbDR.

Proposition .27. *There is a polynomial p such that for every set of Guarded TGDs Σ, letting s_i be the size of* $\mathsf{ISkolemDR}$ *and g_i be the size of* IExbDR *we have $s_i \leq p(g_i)$.*

Proof. Fix Guarded TGDs Σ, let Σ' be the Skolemization of Σ, and let w be the width of Σ.

Given a Guarded TGD σ and a body-full Skolemized TGD σ', we say that σ *covers* σ' if, after applying a renaming to σ', we have:

- the head of σ contains an atom whose Skolemization is the head of σ'
- σ' and σ have the same body atoms

Note that if σ' is function-free, then if σ covers σ' the two must be equal to each other up to renaming.

We first show that for each collection of body-full Skolemized TGDs $\sigma'_1 \dots \sigma'_n$ in $\mathsf{ISkolemDR}$ such that σ'_i share a Skolem function in their head, there is a TGD σ in IExbDR such that the Skolemization of σ covers each σ'_i.

In particular, the number of body-full rules in the output of SkolemDR is bounded by a polynomial in the number of rules in IExbDR.

We prove this by induction on the number of rounds in which the σ'_i are produced. The base case where all σ'_i are Skolemizations of the original Guarded TGDs is obvious.

For the inductive case we have a body-full rule σ'_1 generated in round $r+1$ of the closure, and other body-Skolemized rules $\sigma'_2 \dots \sigma'_n$ generated in earlier rounds. Recall that the SkolemDR inference rule does resolution of a body-full Skolemized TGD λ' with a Skolemized TGD ρ', and that there are two cases. The first is where λ' is function-free and the atom B' in ρ' that gets resolved is a guard for the body: let us call this a *potentially Skolem introducing* (PSI) step, since it may introduce Skolems in the body of the resolvent. The second case is where B' is functional. We call such a rule *Skolem reducing* (SR), since in this case the resolvent will have fewer functional atoms in its bodies than ρ'. It is clear that σ'_1 must be obtained in the following manner: first a PSI step must be applied to a body-full TGD λ'_0 and a function-free ρ'_0, where λ'_0 is generated in an earlier round; this must be followed by at most w SR steps applied to the resolvent: we assume exactly w for simplicity below. Let $\lambda'_1 \dots \lambda'_w$ be the body-full TGDs in these SR steps, with each λ'_i associated with a body atom B_i produced by the PSI step. Then there must be a Skolem function shared by the heads of each λ'_i and λ'_0. By induction there is γ in IExbDR such that the Skolemization of γ covers λ'_0 and also covers $\lambda'_1 \dots \lambda'_w$. Also by induction, the function-free ρ'_0 is produced by ExbDR. In ExbDR we can now resolve γ with ρ'_0. Since γ subsumes each λ'_i, all existential variables will be included.

We now consider a general σ in ISkolemDR, not necessarily body-full. Such a σ must be obtained from a body-full rule in SkolemDR via applying a PSI step and then at most w SR steps. Thus the number of rules in ISkolemDR is at most $w+1$ times the number of body-full rules in ISkolemDR. ◻

Complexity of SkolemDR

From the simulation of SkolemDR by ExbDR given in the previous section, we see that SkolemDR's complexity matches the upper bounds of ExbDR (which matches the known lower bounds in [14]). This can also be proven directly by bounding the number of intermediate rules produced. In particular, we have:

Lemma .28. SkolemDR *terminates in* PTIME *for bounded w, n, a, c,* EXPTIME *for bounded a, and* 2EXPTIME *otherwise.*

PROPERTIES OF HyperDR

Correctness of HyperDR

We show that HyperDR simulates SkolemDR, more formally: any body-full $\tau \in \mathsf{ISkolemDR}$ is also in $\tau \in \mathsf{IHyperDR}$. From this it will follow that the HyperDR algorithm is complete.

The base case is trivial since both algorithms start with the same set of skolemized TGDs.

Assume we (SK-RESOLVE) $\tau, \tau' \in \mathsf{ISkolemDR}(\Sigma)$ to obtain the body-full τ''. By induction τ is in $\mathsf{IHyperDR}(\Sigma)$. We consider two cases:

If τ' is body-full, then also τ' is in $\mathsf{IHyperDR}(\Sigma)$ by induction. Since τ'' is body-full, we can apply (G-HYP) to obtain τ'' in the same manner we applied (SK-RESOLVE).

Now consider the case where τ' is body-non-full. Then τ' must have been the result of a series of (SK-RESOLVE) steps using body-full but non-full $\tau_1, \ldots, \tau_n \in$ ISkolemDR(Σ) and a body-full $\tau'_1 \in$ ISkolemDR(Σ) such that applying (SK-RESOLVE) on τ_1, τ'_1 returns the body non-full rule τ'_2, applying (SK-RESOLVE) on τ_2, τ'_2 the body-non-full τ'_3, and so forth, until finally obtaining $\tau'_{n+1} = \tau'$. Then by induction, $\tau_1, \ldots, \tau_n, \tau'_1 \in$ IHyperDR(Σ). Now we can use (G-HYP) on $\tau_1, \ldots, \tau_n, \tau$ and τ'_1 to obtain τ''.

Complexity of HyperDR

From the simulation of HyperDR by SkolemDR given in the previous section, we see that the rules derived by HyperDR are contained in those derived by SkolemDR. It is then easy to see that also HyperDR matches the known optimal complexity bounds:

Lemma .29. HyperDR *terminates in* PTIME *for bounded* w, n, a, c, EXPTIME *for bounded* a, *and* 2EXPTIME *otherwise.*

OBSERVATIONS CONCERNING ExbDR VS SkolemDR

We mention in the body of the paper that ExbDR can outperform Skolemized approaches like SkolemDR on some examples in the ontology repository. We now give an example of this phenomenon, abstracted from Ontology 389 in the Oxford repository,

`OBO/biological_process_xp_plant_anatomy/2012-10-12`

Consider GTGDs whose Skolemization consists of the following Skolemized rules:

(1) $A_i(x) \to B(x, f_i(x))$ for $1 \leq i \leq n$
(2) $A_i(x) \to P(f_i(x))$ for $1 \leq i \leq n$
(3) $B(x, y) \wedge P(y) \wedge C(y) \to D_j(x)$ for $1 \leq j \leq k$
(4) $P(x) \to Q(x)$

Note that these examples can be expressed in standard ontology languages. Indeed we extracted this pattern from ontology 389 mentioned above. ExbDR derives very little from the original rules, since the existential variable check will prevent resolution of $A_i(x) \to \exists z\, B(x, z) \wedge P(z)$ with $B(x, y) \wedge P(y) \wedge C(y) \to D_j(x)$.

In SkolemDR, we will resolve each of the n rules in the first family with each of the k rules in the third family. Our look-ahead optimization from Section 6 will allow us to see that resolving rules from the second family with rules with the fourth rule is not useful, since Q does not occur in the body of any TGD. But still we see a quadratic advantage for ExbDR over SkolemDR, even with ontologies.

DETAILS ON ALTERNATIVE ALGORITHMS

In the body of the paper we mention alternative algorithms for deciding query answering. The most well-known is via reduction to tree automata, which is implicit in several different methods proposed in [14]: one via Courcelle's theorem [1] and one (specific to atomic query answering), via reduction to satisfiability in the guarded fragment [5]. In the analysis, we consider the following parameters:

- n the number of distinct predicates,
- a the maximal arity of the predicates,
- w_h the maximal head width of any rule, and
- b the number of elements in the base instance.

As we have seen via the chase, if an atomic query is not entailed by an instance I and a set of GTGDs Σ, there is a counterexample with a tree shape. Restricting to a query and dependencies without constants, in this counterexample each node other than the root has at most w_h elements in it, while the root has b elements. Such an instance can be represented as a labelled tree, where each label represents the collection of facts over that chase tree node. The prior technique of [14] amounts to building an automaton for such trees, and checking if it is non empty. Let us ignore the states and transitions of the automaton, along with all the algorithms that need to be performed to check non-emptiness. We consider only the *label set* of trees accepting automaton, which is the set of possible contents of any chase node. For the moment we further ignore the possible values of the root node, focusing only on the potential labels of non-root nodes.

Let us consider first nodes where the elements are guarded, as would happen if the GTGDs had a single head. For each relation there are then a total of at most a elements in the node, and each of these are chosen from the base instance or as nulls. For the moment, we do not distinguish between the nulls. There are $n \cdot b^a$ facts over a signature with n predicates each with a places, where the arguments come from a set of size b, thus a crude upper bound for the number of labels is:

$$2^{n \cdot b^a}$$