University of Oxford

Part C Project Report

Rewriting Conjunctive Queries Under Guarded TGDs

Ryosuke Kondo

Supervised by
Prof. Michael Benedikt

Trinity Term 2023

Abstract

The class of Guarded Tuple-Generating Dependencies (GTGDs) is an expressive class of first-order formulae for which conjunctive query answering is decidable. Even though the problem is 2EXPTIME-complete in general, it has been shown that, by essentially performing enough pre-computation, one can *rewrite* GTGD conjunctive query answering problem to a Datalog query, which can be run in time polynomial to the size of the input database. Nevertheless, no work has implemented the rewriting algorithm for general conjunctive queries under GTGDs. By utilising the recent implementation for rewriting GTGDs over atomic queries, we revisit the theory of chases, derive a rewriting algorithm for general queries and discuss its implementation and further optimisations.

Contents

Chapter 1

Introduction

1.1 Background

Query answering under data integration rules is one of the central problems in knowledge representation and reasoning. To give an idea of the problem, consider the following *relational database*, a collection of tuples in relations.

$$\mathcal{D} = \{U(a), R(b,c), S(d,d)\}$$

Suppose that, as domain knowledge, we know that

- for any X, if $U(X)$ is a relation then there should be some value Z with $R(X,Z)$, and
- for any X, Y, $R(X,Y)$ should imply $S(X,X)$.

The database $\mathcal{D}$ is *incomplete*, as it does not satisfy the abovementioned constraints. Nevertheless, we could still ask a question of the form: "*If* the database had been fixed by adding some facts not yet recorded by the information system at the moment, what values v would satisfy $S(v,v)$?" In the example above, we can say that all of a, b, and d become answers to this question regardless of how we fix $\mathcal{D}$.

This type of problem typically arises when data sources are distributed, such as in the Semantic Web, and the querying party needs to integrate the data to make sense of possibly incomplete data.

A class of logical formulas known as Tuple Generating Dependencies (TGDs), which are of the form $\forall \vec{x}.\beta \rightarrow \exists \vec{y}.\ \eta$, could describe data integrity rules. In the example above, the two rules would be written as $\forall X.\ U(X) \rightarrow \exists Z.\ R(X,Z)$ and $\forall X, Y.\ R(X,Y) \rightarrow S(X,X)$.

Unfortunately, query answering under general TGDs is undecidable [3]. A line of work, including [6], identified *Guarded TGDs* (GTGDs) as a syntactically restricted subclass of TGDs that leaves query-answering decidable yet much more expressive than description logics used for ontological reasoning.

The first result that opened up the possibility towards practical query answering is shown in [2], which states that we can compute a *Datalog rewriting* of (frontier) guarded TGDs and a conjunctive query. Roughly speaking, a Datalog rewriting is a set of existential-free TGDs that gives the same answer as the original query. It is well-known that a fixed Datalog program can be run in a polynomial time on a database.

More recently, [4] implemented the system for answering *atomic* queries over GTGDs. Their algorithm takes the GTGD rule set Σ as an input and outputs a Datalog program that derives all tuples which could have been derived using Σ. We call this program an *atomic rewriting* of Σ to distinguish from rewritings of general conjunctive queries.

To our knowledge, however, no work has yet implemented a query-answering procedure for general conjunctive queries under GTGDs. This project aims to bring the theory closer to implementations using atomic rewritings produced by [4]. Ultimately, we provide a working prototype of a query-answering system that works with GTGDs.

1.2 Contribution of This Work

The primary theoretical contribution of this work is the development of a concrete Datalog rewriting algorithm. On our way, we introduce a variant of chase which we call *shortcutting chase tree* and develop some theory concerning query satisfaction within the chase structure. We then apply the theory to derive a Datalog rewriting, demonstrating room for further optimisations.

In addition to the theoretical work, we provide the first implementation of the GTGD rewriting algorithm in Java, incorporating some optimisations we will have discussed.

1.3 Outline of This Report

In Chapter 2, we review basic terminologies and results that we use throughout the report.

In Chapter 3, we revisit *tree-like chase proofs* and introduce the notion of *shortcutting chase trees*. We then analyse how answers to a conjunctive query are embedded into the shortcutting chase tree, emphasising its relationship to the connectedness of the query. At the end of the chapter, we apply our observations to derive a query-answering procedure that operates over shortcutting chase trees.

In Chapter 4, we use the observations from Chapter 3 to plan towards a Datalog rewriting procedure. We will see that the query answering procedure from Chapter 3 can be used to derive a rewriting, although it is inefficient as is. We close the chapter with optimisations of the algorithm so obtained.

We briefly review our implementation in Chapter 5. Finally, we conclude the report with an overview and discuss some weaknesses of the algorithm to be improved in future studies.

Chapter 2

Preliminaries

This chapter introduces terminologies and notations, which we will use throughout the report.

2.1 Formulas

We fix countably infinite collections of *constants*, *variables* and *nulls*. Nulls are similar to variables but play a similar role as Skolem constants (at the beginning of Chapter 3, we will see examples illustrating the use of nulls). A *term* is either a constant, a variable or a null.

We also fix a countably infinite set of *predicates*. A predicate P has its associated arity $\mathrm{Arity}(P) \in \mathbb{N}_{\geq 0}$. An *atom* is a string of the form $P(t_1, \ldots, t_{\mathrm{Arity}(P)})$, where each t_i is a term. A *fact* is an atom not containing a variable, and a *base fact* is a fact not containing nulls. We write $\mathrm{Terms}(S)$, $\mathrm{Consts}(S)$, $\mathrm{Vars}(S)$ and $\mathrm{Nulls}(S)$ for the sets of terms, constants, variables and nulls appearing in an object S (such as a conjunction of atoms).

A *tuple-generating dependency (TGD)* is a formula of the form $\tau = \forall \vec{x}.\ \beta \to \exists \vec{y}.\ \eta$, where β and η are conjunctions of null-free atoms containing variables from $\vec{x}$ and $\vec{x} \cup \vec{y}$ respectively. The conjunctions β and η are called the *body* and the *head* of τ, respectively. The set $\mathrm{Vars}(\beta) \cap \mathrm{Vars}(\eta)$ of variables appearing in both the body and the head of τ is called the *frontier* of τ.

A TGD $\forall \vec{x}.\ \beta \to \exists \vec{y}.\ \eta$ without an existential variable (i.e. $\vec{y} = \emptyset$) is said to be *full*. Full TGDs are also called *Datalog* rules. Conversely, a non-full rule is said to be *existential*. A finite set of Datalog rules is called a *Datalog program*.

We say that a conjunction $A_1 \wedge \ldots \wedge A_n$ of atoms is *guarded* if there is an atom A_i with $\mathrm{Vars}(A_i) = \mathrm{Vars}(A_1 \wedge \ldots \wedge A_n)$, and that a TGD is a *guarded TGD (GTGD)* if its body β is guarded. A GTGD $\forall \vec{x}.\ \beta \to \exists \vec{y}.\ \eta$ is *single-headed* if the head η only contains a single atom.

Remark 2.1. Note that a full rule $\forall \vec{x}.\ \beta \to H_1 \wedge \ldots \wedge H_n$ can always be split into n rules $\forall \vec{x}.\ \beta \to H_i$ for $1 \leq i \leq n$. We will, therefore, implicitly treat all full rules as single-headed rules.

Remark 2.2. We can also split a non-single-headed existential rule $\tau = \forall \vec{x}.\ \beta \rightarrow \exists \vec{y}.\ H_1 \wedge \ldots \wedge H_n$, but only if we introduce a new predicate. More precisely, suppose that $\vec{z} \subseteq \vec{x}$ is the frontier of τ, then we can turn τ into two rules

- $\forall \vec{x}.\ \beta \rightarrow \exists \vec{y}.\ I(\vec{z}, \vec{y})$, and
- $\forall \vec{z}, \vec{y}.\ I(\vec{z}, \vec{y}) \rightarrow H_1 \wedge \ldots \wedge H_n$, which is full.

In general, a set Σ of GTGDs can be turned into a set of single-headed GTGDs if we introduce an intermediate predicate for each existential rule in Σ. $\square$

A *conjunctive query* is a formula of the form $\exists z_1, \ldots, z_n.\ A_1 \wedge \ldots \wedge A_m$, where $n \geq 0$, $m \geq 1$, and each A_i is an atom. We write $\mathrm{FV}(Q)$ for the set of free variables in Q. A conjunctive query is said to be *Boolean* if $\mathrm{FV}(Q) = \emptyset$.

2.2 Database Instances and Homomorphisms

We now turn our attention to a representation of a relational database. A *(database) instance* is a collection of facts. A base (database) instance is a collection of base facts. We often regard a database instance as a single conjunction of facts. For example, we identify an instance $\{R(c_1, c_3), T(c_2, c_3, c_3)\}$ with a conjunction $R(c_1, c_3) \wedge T(c_2, c_3, c_3)$.

We use *homomorphisms* to describe how an instance satisfies a conjunctive query.

An *instance homomorphism* $\sigma : \mathcal{D} \rightarrow \mathcal{D}'$ from an instance $\mathcal{D}$ to another instance $\mathcal{D}'$ is an "embedding" of $\mathcal{D}$ into $\mathcal{D}'$. More precisely, σ is a mapping $\sigma : \mathrm{Nulls}(\mathcal{D}) \rightarrow \mathrm{Terms}(\mathcal{D}')$, such that for every fact $F \in \mathcal{D}$, $\sigma(F) \in \mathcal{D}'$.

Similarly, a *query homomorphism* $\sigma : Q \rightarrow \mathcal{D}$ from a conjunctive query $Q = \exists \vec{z}. \bigwedge_{j \in J} A_j$ to an instance $\mathcal{D}$ is a mapping $\sigma : \mathrm{Vars}(Q) \rightarrow \mathrm{Terms}(\mathcal{D})$ such that $\sigma(A_j) \in \mathcal{D}$ for each $j \in J$.

We say that an instance $\mathcal{D}$ *satisfies* a Boolean conjunctive query Q, written $\mathcal{D} \models Q$, if there exists a query homomorphism $\sigma : Q \rightarrow \mathcal{D}$. We also say that the instance $\mathcal{D}$ together with a set Σ of TGDs *entails* Q, written $\mathcal{D} \cup \Sigma \models Q$, if for every extension $\mathcal{D}_{\mathrm{ext}}$ of $\mathcal{D}$ satisfying Σ, $\mathcal{D}_{\mathrm{ext}}$ satisfies Q. Note that this definition of query satisfaction and entailment is consistent with the ordinary model-theoretic interpretation of $\models$ relation.

2.3 Datalog Saturation

Consider the following problem:

Problem 2.3. Given an existential-free conjunctive query Q, an input database $\mathcal{D}$ and a Datalog program P, find all substitutions (each of which is an *answer* to the query) $\alpha : \mathrm{FV}(Q) \rightarrow \mathrm{Consts}(\mathcal{D})$ such that $\mathcal{D} \cup P \models \alpha(Q)$.

Several algorithms for this problem are known, and [1] compared a few classic algorithms. One of the simplest is the *Naive Evaluation* (Algorithm 1). Roughly speaking, we keep adding tuples produced by rules in P to the input database until we reach a fix-point. We then evaluate Q, which is merely a join query, on the populated database to obtain all answers.

Algorithm 1 Answering Datalog query using Naive Evaluation

procedure DatalogSaturate(Datalog program P, input database $\mathcal{D}$)
  $\mathcal{D}_{\text{current}} \leftarrow \mathcal{D}$
  while true **do**
    $\mathcal{D}_{\text{next}} \leftarrow \mathcal{D}_{\text{current}}$
    for each $r \in P$ **do**
      for each $\sigma \in$ result of matching the body of r on $\mathcal{D}_{\text{current}}$ **do**
        add all head atoms of r substituted with σ to $\mathcal{D}_{\text{next}}$
      end for
    end for

    if $\mathcal{D}_{\text{next}} \neq \mathcal{D}_{\text{current}}$ **then**
      $\mathcal{D}_{\text{current}} \leftarrow \mathcal{D}_{\text{next}}$
    else
      return $\mathcal{D}_{\text{current}}$
    end if
  end while
end procedure

procedure AnswerDatalogQuery(P, $\mathcal{D}$, $\exists$-free query Q)
  return result of running the join query Q on DatalogSaturate($P, \mathcal{D}$)
end procedure

We call the database instance DatalogSaturate($P, \mathcal{D}$) the *Datalog saturation of* $\mathcal{D}$ *with* P. Intuitively, a Datalog saturation results from *supplementing* the input instance by adding all (but nothing other than) facts derivable from the original instance.

For a general set Σ of GTGDs, we can compute a Datalog program, which we call an *atomic rewriting*, that gives the same answers as Σ for atomic queries. More precisely,

Definition 2.4. Let Σ be a finite set of GTGDs. We say that a Datalog program P is an *atomic rewriting* of Σ if for every instance $\mathcal{D}$ and every fact F, $\mathcal{D} \cup \Sigma \models F$ if and only if $\mathcal{D} \cup P \models F$.

A recent work [4] implemented an algorithm called *Guarded-saturation* for computing an atomic rewriting of an arbitrary GTGD set. For this report, we fix an implementation in Guarded-saturation and call its output *the* atomic rewriting ARew(Σ) of Σ.

2.4 Problem Formulation

Our ultimate goal is to answer the following problem.

Problem 2.5 (GTGDs-CQ Answering)**.** Given a finite set Σ of GTGDs, a conjunctive query $Q = \exists \vec{z}. \bigwedge_{j \in J} A_j$ and a base instance $\mathcal{D}$, what are the *answers* to Q under Σ and $\mathcal{D}$, i.e. substitutions $\alpha : \mathrm{FV}(Q) \to \mathrm{Consts}(\mathcal{D})$ such that $\mathcal{D} \cup \Sigma \models \alpha(Q)$?

It is known that a conjunctive query under GTGDs admits *Datalog rewritings* [2]. These Datalog programs with a designated *goal atom* give the same set of answers as the original rule-query pair when run on any base instance. More precisely:

Definition 2.6. A *Datalog rewriting* of a GTGDs-CQ pair (Σ, Q) is a Datalog program $\Sigma_{\mathrm{Datalog}}$ together with an atomic query Q_{atomic} with $\mathrm{FV}(Q_{\mathrm{atomic}}) = \mathrm{FV}(Q)$, such that for every base instance $\mathcal{D}$ and a substitution $\alpha : \mathrm{FV}(Q) \to \mathrm{Consts}(\mathcal{D})$ for Q,

$$\mathcal{D} \cup \Sigma \models \alpha(Q) \textit{ if and only if } \mathcal{D} \cup \Sigma_{\mathrm{Datalog}} \models \alpha(Q_{\mathrm{atomic}}).$$

We call Q_{atomic} the *goal atom* in the Datalog rewriting. □

In order to answer Theorem 2.5, it is desirable to run a Datalog rewriting of the pair (Σ, Q) on $\mathcal{D}$ instead of directly verifying $\mathcal{D} \cup \Sigma \models \alpha(Q)$ for every possible substitution $\alpha : \mathrm{FV}(Q) \to \mathrm{Consts}(\mathcal{D})$, since running a fixed Datalog program on $\mathcal{D}$ only takes time polynomial in the size of $\mathcal{D}$ [7].

This project aims to derive an algorithm that produces Datalog rewritings for arbitrary GTGDs-CQ pairs. We achieve this using the atomic rewriting produced by the Guarded-saturation algorithm.

Chapter 3

Characterising Query Entailment under GTGDs

3.1 Tree-Like Chase Proofs

In the presence of existential rules, an object known as the *chase* represents a canonical *completion* of a database concerning some data integration rules. Chases are constructed similarly to Datalog saturations by adding tuples produced by rules, except we need to replace existential variables with *nulls*, which represent Skolem constants [8].

As with Datalog saturations, the chase of an instance $\mathcal{D}$ gives all possible answers $\alpha : \mathrm{FV}(Q) \to \mathrm{Consts}(\mathcal{D})$ to a conjunctive query $Q = \exists \vec{z}. \bigwedge_{j \in J} A_j$. That is, we can find all answers to Q by evaluating a join query $\bigwedge_{j \in J} A_j$ on the chase and projecting $\vec{z}$ away.

However, with recursive rules[1] such as $R(x, y) \to \exists z.\ R(y, z)$, the chase procedure may have to be continued indefinitely, producing an infinite chase. Moreover, as one keeps populating the database instance with tuples regardless of whether they eventually affect query output, the chase lacks a structure with which we can reason about query entailments.

To deal with this issue, [4] introduced the notion of *tree-like chase proofs*, which essentially capture instants of ongoing chase processes. To precisely describe this, we will use the following terminology.

Definition 3.1. A *chase tree* is a pair of a (potentially infinite) rooted directed tree T together with a function μ mapping each vertex $v \in T$ to a set of facts (possibly with nulls). We usually refer to the instance $\mu(v)$ as a *bag of facts at* v to distinguish it from the input base instance.

We will often identify a chase tree (T, μ) with the set $\bigcup_{v \in T} \mu(v)$ of facts in it. $\square$

In this section, we focus on finite chase trees. The following definition incorporates ideas from [4] and [9].

[1]A TGD is recursive if a predicate appears both in the body and the head.

Definition 3.2. Given a set Σ of single-headed GTGDs and a base instance $\mathcal{D}$, a *tree-like chase proof over* Σ *from* $\mathcal{D}$ is a finite sequence of chase trees $C_1, \ldots, C_n$ such that

- $C_1 = (T, \mu)$ is a chase tree with a single vertex v together with $\mu(v) = \mathcal{D}$.
- for each $i < n$, C_{i+1} is obtained from $C_i = (T, \mu)$ by applying one of the following transformation steps.
  - *a chase step* from a vertex $v \in T$ with $\tau = \forall \vec{x}.\ \beta \to \exists \vec{y}.\ H$ and a substitution σ mapping $\vec{x}$ to terms in $\mu(v)$, provided that $\sigma(\beta) \subseteq \mu(v)$. The result (T', μ') of this step depends on whether τ is full, i.e. if $|\vec{y}| = 0$:
    * If τ is full, we simply add the head atom H of τ to the instance at v. That is, we keep $T' = T$, and set $\mu'(v) = \mu(v) \cup \sigma(H)$ while maintaining μ and μ' equal on $T \setminus \{v\}$.
    * If τ is existential, we create a fresh child of v, put the generated tuple in it and *inherit* facts from the parent bag. Formally, we first extend σ to a substitution σ' that maps variables in $\vec{y}$ to fresh nulls. We define the bag I containing *inherited facts* as

      $$I = \{F \in \mu(v) \mid \mathrm{Terms}(F) \subseteq \mathrm{Terms}(\sigma'(H)) \cup \mathrm{Consts}(\Sigma)\}.$$

      Finally, we prepare a fresh vertex c, form T' by making c a child of v, and extend μ to μ' with $\mu'(c) = \{\sigma'(H)\} \cup I$.
  - *a propagation step* from a vertex $v \in T$ to an ancestor $a \in T$ of v with a fact $F \in \mu(v)$, provided that $F \notin \mu(a)$ and $\mathrm{Terms}(F) \subseteq \mathrm{Terms}(\mu(a)) \cup \mathrm{Consts}(\Sigma)$. The resulting chase tree is (T, μ'), where μ' agrees with μ everywhere except at a, and $\mu'(a) = \mu(a) \cup \{F\}$.

$\square$

A tree-like chase proof over Σ from $\mathcal{D}$ represents a process of supplementing $\mathcal{D}$ according to Σ. We can derive new tuples using a chase step, stepping off to a child node when firing an existential rule. With a propagation step, we can retrieve a fact derived at a descendant node to its ancestor.

Example 3.3. Suppose that Σ contains the following single-headed GTGDs, where c_1 is a constant.

$$\begin{array}{rcrcl}
\tau_1 & = & \forall x_1, x_2.\ A(x_1, x_2) & \to & \exists y.\ B(x_1, y) \\
\tau_2 & = & \forall x_1, x_2.\ B(x_1, x_2) & \to & C(x_1, c_1) \wedge A(x_2, c_1) \\
\tau_3 & = & \forall x_1, x_2.\ A(x_1, x_2) \wedge C(x_1, x_2) & \to & \exists y.\ D(x_2, y)
\end{array}$$

If we start with an instance $\mathcal{D} = \{A(c_4, c_1)\}$, we can perform a chase step with τ_1, then chase with τ_2, propagate a fact $C(c_4, c_1)$ to the root and finally fire τ_3. We illustrate the process in Figure 3.1. $\square$

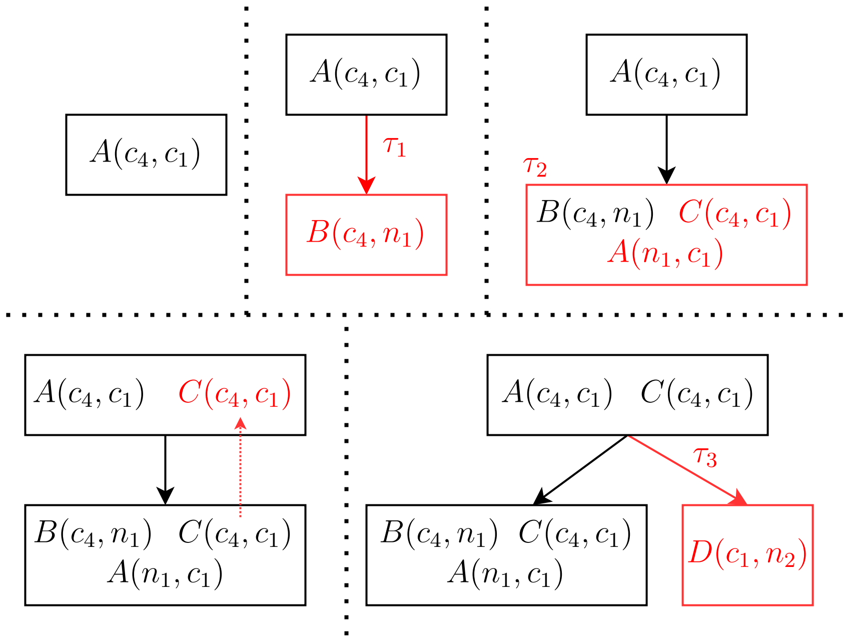

Figure 3.1: A tree-like chase proof with rules in Theorem 3.3. At each stage, elements newly added to the chase tree are highlighted in red.

Note that we only propagate or inherit facts whose terms appear in the target node. Intuitively, such a restriction can be justified because the guardedness of rules in Σ prohibits us from *combining* unrelated terms into new tuples. We cannot hope to fire an existential rule from a node $a \in T$ in a meaningfully novel manner even if we transfer to a a fact with terms not already present in a. That is, if we propagated a fact F from v to a and could fire a rule (τ, σ) on a *using* F, we could have fired (τ, σ) on v in the first place, provided that v inherited enough facts related to F from a[2].

The restriction mentioned above makes the local structure of tree-like chase proofs *small* in the following sense.

Definition 3.4. Let Σ be a set of GTGDs and $K \in \mathbb{N}$. We say that a bag $\mathcal{B}$ of facts is (K, Σ)-*small* when $|\mathrm{Terms}(\mathcal{B}) \setminus \mathrm{Consts}(\Sigma)| \leq K$.

Proposition 3.5 (Tree-width of chase proofs)**.** *Let Σ be a set of* single-headed *GTGDs, $(T_1, \mu_1), \ldots, (T_n, \mu_n)$ a tree-like chase proof over Σ from $\mathcal{D}$. Let K be the maximum arity of predicates that appear in $\mathcal{D} \cup \Sigma$. Then for every $1 \leq i \leq n$ and every non-root vertex v of T_i, $\mu_i(v)$ is (K, Σ)-small. In particular, if $\mathcal{D}$ is (K, Σ)-small, all bags in (T_i, μ_i) are (K, Σ)-small.*

Proof. By induction on i. The base case $i = 1$ is vacuous. We look at the rule used to derive (T_{i+1}, μ_{i+1}) from (T_i, μ_i) and examine the bag at the modified

[2]In fact, a might have more facts with terms in F than v does due to propagation from cousin nodes of v. In that case, we could "re-derive" a cousin node v' from a in the same way as how we had derived v, and v' should have as many facts related to an isomorphic copy of F as a does.

node.

- If the last step was a chase step with a full rule or a propagation step, no term set has been modified for any bag in the chase tree.
- If the last step was a chase step from a vertex $v \in T_i$ with an existential rule $\forall \vec{x}.\ \beta \to \exists \vec{y}.\ H$ and a substitution σ, we have added a new node c under v. As $|\mathrm{Terms}(\sigma'(H))| \leq K$ and the inherited bag I has $\mathrm{Terms}(I) \subseteq \mathrm{Terms}(\sigma'(H)) \cup \mathrm{Consts}(\Sigma)$,

$$\begin{aligned}|\mathrm{Terms}(\mu_{i+1}(c)) \setminus \mathrm{Consts}(\Sigma)| &= |\mathrm{Terms}(\{\sigma'(H)\} \cup I) \setminus \mathrm{Consts}(\Sigma)| \\ &\leq K.\end{aligned}$$

When $\mathcal{D}$ is (K, Σ)-small, the result extends to the root node since no chase transformation modifies the term set of the root node. □

At the same time, as long as conjunctive queries are concerned, tree-like chase proofs prove everything we can reason about the input database $\mathcal{D}$.

Proposition 3.6. *For any set Σ of single-headed GTGDs, an instance $\mathcal{D}$ and a Boolean conjunctive query Q, there exists a tree-like chase proof $C_1, \ldots, C_n$ over Σ from $\mathcal{D}$ such that $C_n \models Q$ if and only if $\mathcal{D} \cup \Sigma \models Q$.*

Proof (sketch).
($\Longrightarrow$, "soundness" of chase proofs): Suppose that $(T_1, \mu_1), \ldots, (T_n, \mu_n)$ is a tree-like chase proof over Σ from $\mathcal{D}$. For each $1 \leq i \leq n$, let $I_i = \bigcup_{v \in T_n} \mu_n(v)$ be the instance containing all facts in (T_i, μ_i). The sequence $I_1, \ldots, I_n$ is known as a *chase sequence*, and [8] showed that Q follows from $\mathcal{D}$ if there is a chase sequence ending with $I_n \models Q$.

($\Longleftarrow$, "completeness" of chase proofs): This direction is proven in [9, Proposition 2.6.9]. The proof therein decomposes an ordinary chase sequence into a tree-like structure, and the decomposition is performed exactly as in Theorem 3.2. □

3.2 Shortcutting Chase Trees

Tree-like chase proof is a powerful tool to reason about a fragment of a chase, but unlike a full chase, they do not constitute a model of $\mathcal{D} \cup \Sigma$. Therefore, we wish to construct a model that retains a tree-like structure.

Notice that during a chase proof, we only ever fire an existential rule from a node v in the hope of either

- retrieving back a fact to v to accumulate as much facts as possible to v, or
- finding an existential witness to the query variable in the subtree of v.

We can immediately achieve the former objective by computing the Datalog saturation of the bag at v using the atomic rewriting $\mathrm{ARew}(\Sigma)$ of Σ. This observation motivates the following definition.

Definition 3.7 (Shortcutting Chase Trees)**.** Let Σ be a finite set of single-headed GTGDs and $\mathcal{D}$ an instance. We inductively define an infinite sequence $(T_0, \mu_0), (T_1, \mu_1), \ldots$ of chase trees as follows:

- (T_0, μ_0) is a chase tree with only the root vertex r together with

$$\mu_0(r) = \textsc{DatalogSaturate}(\textsc{ARew}(\Sigma)\,, \mathcal{D}).$$

- For the inductive step, (T_{i+1}, μ_{i+1}) is constructed by shortcut-chasing all leaves in the previous chase tree (T_i, μ_i).

  More precisely, let L_i be the set of leaf nodes in T_i. For each $l \in L_i$, an existential rule $\tau = \forall \vec{x}.\ \beta \to \exists \vec{y}.\ H$ and a substitution σ mapping $\vec{x}$ into $\mathrm{Terms}(\mu_i(l))$ such that $\sigma(\beta) \subseteq \mu_i(l)$, define

  - σ' as an extension of σ that maps all variables in $\vec{y}$ to fresh nulls
  - the bag $B_{l,\tau,\sigma}$ of facts *inherited from* $\mu_i(l)$ *through* τ *and* σ as

$$B_{l,\tau,\sigma} = \{F \in \mu_i(l) \mid \mathrm{Terms}(f) \subseteq \mathrm{Terms}(\sigma'(H)) \cup \mathrm{Consts}(\Sigma)\}$$

  We construct T_{i+1} as an extension of T_i by adding a vertex $c_{l,\tau,\sigma}$ as a child of l for each such $l \in L_i$, τ and σ. We extend μ_i to μ_{i+1} by setting

$$\mu_{i+1}(c_{l,\tau,\sigma}) = \textsc{DatalogSaturate}(\textsc{ARew}(\Sigma)\,, \{\sigma'(H)\} \cup B_{l,\tau,\sigma}).$$

Finally, the *shortcutting chase tree* $\mathrm{SCTree}(\mathcal{D}, \Sigma)$ *of* $\mathcal{D}$ *over* Σ is a chase tree defined as the limit $(\bigcup_{i=0}^{\infty} T_i, \bigcup_{i=0}^{\infty} \mu_i)$ of the sequence defined above. $\square$

Example 3.8. Consider the rules in Theorem 3.3. According to an implementation in [5], the set Σ' of the following two rules is an atomic rewriting of Σ.

$$\begin{array}{rcrcl} \tau_1' & = & \forall x_1, x_2.\ B(x_1, x_2) & \to & C(x_1, c_1) \land A(x_2, c_1) \\ \tau_2' & = & \forall x_1, x_2.\ A(x_1, x_2) & \to & C(x_1, c_1) \end{array}$$

A first few layers of $\mathrm{SCTree}(\{A(c_4, c_1)\}, \Sigma)$ is illustrated in Figure 3.2. The chase tree is continued indefinitely by chasing with existential rules and Datalog-saturating each layer with Σ'. $\square$

The structure of a shortcutting chase tree is very similar to that of a chase proof.

Proposition 3.9. *Let Σ be a set of single-headed GTGDs and $\mathcal{D}$ an instance. Let K be the maximum arity of predicates that appear in $\mathcal{D} \cup \Sigma$. Then every bag of facts at a non-root node of* $\mathrm{SCTree}(\mathcal{D}, \Sigma)$ *is (K, Σ)-small. In particular, if $\mathcal{D}$ is (K, Σ)-small, all bags in* $\mathrm{SCTree}(\mathcal{D}, \Sigma)$ *are (K, Σ)-small.*

Proof. By the same analysis as in the proof of Theorem 3.5. $\square$

Expectedly, $\mathrm{SCTree}(\mathcal{D}, \Sigma)$ is a universal model for $\mathcal{D} \cup \Sigma$, since any finite subtree of $\mathrm{SCTree}(\mathcal{D}, \Sigma)$ homomorphically embeds into some chase proof.

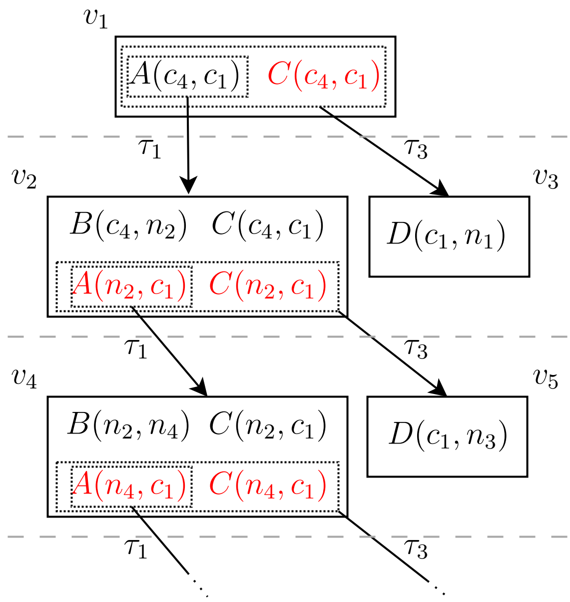

Figure 3.2: A shortcutting chase tree over an instance $\{A(c_4, c_1)\}$ with rules in Theorem 3.3. The facts highlighted in red are obtained by Datalog-saturating the inherited bag with an atomic rewriting, and dotted boxes indicate substituted bodies used to fire an existential rule. Notice how the subtree rooted at v_1 can be obtained as a shortcutting chase of the bag at v_1, as remarked in Theorem 3.12.

Lemma 3.10. *Let $\mathcal{D}$ be an instance and Σ a finite set of single-headed GTGDs. Then for any finite rooted subtree (T, μ) of* $\mathrm{SCTree}(\mathcal{D}, \Sigma)$*, there exists a tree-like chase proof $C_1, \ldots, C_n$ that admits an instance homomorphism $\sigma : (T, \mu) \to C_n$.*

Proof. By induction on the structure of (T, μ).

The base case is $T = \{r\}$, where r is the root of $\mathrm{SCTree}(\mathcal{D}, \Sigma)$. By Theorem 3.6, we can construct a chase-proof $C_1, \ldots, C_n$ that aggregates all provable base facts to the root bag.

For the inductive part, take a finite rooted subtree (T, μ) and a leaf l of T, and suppose that $T \setminus \{l\}$ can be homomorphically mapped into the last chase tree in a proof $C_1, \ldots, C_n$. Let (τ, σ) be a pair of a rule and a substitution used to derive l.

We extend the proof $C_1, \ldots, C_n$ by applying a chase step with (τ, σ) to create a child node $c \in C_{n+1}$. By Theorem 3.6, we can derive all facts whose terms appear in the bag c. We have now obtained a chase proof $C_1, \ldots, C_{n+m}$ with c saturated, so (T, μ) can be homomorphically mapped into C_{n+m}. $\square$

Theorem 3.11. *For a set Σ of single-headed GTGDs, an instance $\mathcal{D}$ and a Boolean conjunctive query $Q = \exists \vec{z}. \bigwedge_{j \in J} A_j$, $\mathcal{D} \cup \Sigma \models Q$ if and only if* $\mathrm{SCTree}(\mathcal{D}, \Sigma) \models Q$.

Proof.

($\Longrightarrow$): Suppose $\mathcal{D} \cup \Sigma \models Q$. By Theorem 3.6, there exists a tree-like chase proof $(T_1, \mu_1), \ldots, (T_n, \mu_n)$ of Q from $\mathcal{D}$ over Σ. By induction on $1 \leq i \leq n$ and by the construction of $\mathrm{SCTree}(\mathcal{D}, \Sigma)$, we can embed each (T_i, μ_i) into $\mathrm{SCTree}(\mathcal{D}, \Sigma)$. Since $(T_n, \mu_n) \models Q$ and (T_n, μ_n) embeds into $\mathrm{SCTree}(\mathcal{D}, \Sigma)$, $\mathrm{SCTree}(\mathcal{D}, \Sigma) \models Q$.

($\Longleftarrow$): Suppose $\mathrm{SCTree}(\mathcal{D}, \Sigma) \models Q$. Then there exists a homomorphism $\sigma : Q \to \mathrm{SCTree}(\mathcal{D}, \Sigma)$.

We may pick a finite rooted subtree (T, μ) of $\mathrm{SCTree}(\mathcal{D}, \Sigma)$ such that σ restricts to $\sigma : Q \to (T, \mu)$. To do so, for each $j \in J$, let $V_j \in \mathrm{SCTree}(\mathcal{D}, \Sigma)$ be a set of vertices whose bags contain the fact $\sigma(A_j)$. Choose $v_j \in V_j$ for each $j \in J$, and let T_j be the set of all ancestors of v_j. Finally, let $T = \bigcup_{j \in J} T_j$.

By applying Theorem 3.10 to (T, μ), there is a chase proof of Q from $\mathcal{D}$ under Σ. By soundness of chase proofs (Theorem 3.6, $\Longrightarrow$) we are done. □

Remark 3.12. A shortcutting tree chase has a *corecursive* structure: If we write μ for the bag assignment function of $\mathrm{SCTree}(\mathcal{D}, \Sigma)$, then for any vertex $v \in \mathrm{SCTree}(\mathcal{D}, \Sigma)$, the subtree T_v of all descendants (including v itself) of v can be written as $T_v = \mathrm{SCTree}(\mu(v), \Sigma)$.

3.3 Query Satisfaction in Shortcutting Chase Trees

We now discuss how the structure of the query constrains the structure of query homomorphisms into the shortcutting chase tree. By the end of this chapter, we will have derived a recursive query answering procedure, whose recursive structure will be exploited in Chapter 4 to compute a Datalog rewriting.

A key observation is that, under a certain condition, a *connected* set of variables produces a connected homomorphic image in $\mathrm{SCTree}(\mathcal{D}, \Sigma)$. To make this intuition precise, we introduce the following terminology.

Definition 3.13. Given a conjunctive query $Q = \exists \vec{z}. \bigwedge_{j \in J} A_j$, we say that

- two variables z_1, z_2 bound in Q are *Q-adjacent* if some atom A_j in Q contains both z_1 and z_2
- a set Z of variables bound in Q are *Q-connected* if for each pair $z_1, z_2 \in Z$ of variables, there is a finite sequence $x_1, \ldots, x_n$ of variables in Z such that
  - $x_1 = z_1$ and $x_n = z_2$
  - for each $1 \leq i < n$, x_i and x_{i+1} are Q-adjacent

For a subset Z of $\vec{z}$, a *Q-connected component of Z* is a $\subseteq$-maximal Q-connected nonempty subset of Z.

Finally, Q is a *connected* conjunctive query if $\vec{z}$ is Q-connected. □

A set Z of bound variables is Q-connected if Z is connected in the hypergraph corresponding to the structure of Q (with Z as the vertex set and atoms in Q as the hyperedges), as illustrated in the following example.

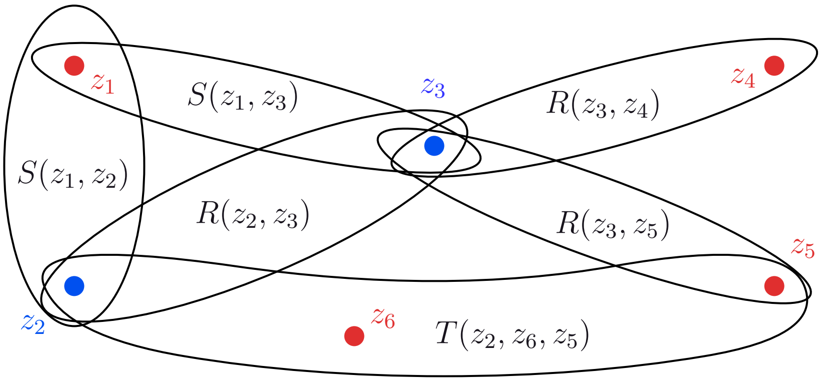

Figure 3.3: The hypergraph corresponding to Q in Theorem 3.14.

Example 3.14. Consider a conjunctive query

$$\begin{aligned} Q = \exists z_1, z_2, z_3, z_4, z_5, z_6.\ & S(z_1, z_2) \wedge S(z_1, z_3) \wedge R(z_2, z_3) \\ & \wedge R(z_3, z_4) \wedge R(z_3, z_5) \wedge T(z_2, z_6, z_5). \end{aligned}$$

Then $\{z_1, z_3, z_4\}$ and $\{z_2, z_3, z_5\}$ are both Q-connected, but $\{z_1, z_2, z_4\}$ and $\{z_4, z_5\}$ are not. Q-connected components of $\{z_1, z_4, z_6, z_5\}$ are $\{z_1\}$, $\{z_4\}$ and $\{z_5, z_6\}$. Q is a connected Boolean query. □

We are ready to state the observation.

Proposition 3.15. *Let* $\mathrm{SCTree}(\mathcal{D}, \Sigma)$ *be a shortcutting chase tree over an instance* $\mathcal{D}$. *Then for any* $t \in \mathrm{Terms}(\mathrm{SCTree}(\mathcal{D}, \Sigma)) \setminus \mathrm{Consts}(\Sigma)$, *the set* V_t *of vertices in* $\mathrm{SCTree}(\mathcal{D}, \Sigma)$ *that contain* t *forms a rooted subtree of* $\mathrm{SCTree}(\mathcal{D}, \Sigma)$.

Proof. Since $t \notin \mathrm{Consts}(\Sigma)$, t appears in a node only if it is

- inherited from the parent node
- a term in $\mathcal{D}$
- a null introduced at the node

Since no null is introduced at two different nodes, V_t must be a rooted subtree of $\mathrm{SCTree}(\mathcal{D}, \Sigma)$. □

Definition 3.16. A query homomorphism $\sigma : Q \to \mathrm{SCTree}(\mathcal{D}, \Sigma)$ is said to be $\mathrm{Consts}(\Sigma)$-free if $\mathrm{range}(\sigma) \cap \mathrm{Consts}(\Sigma) = \emptyset$.

Lemma 3.17 (*Homomorphic image of connected variables is connected*)**.** *Let* $Q = \exists \vec{z}. \bigwedge_{j \in J} A_j$ *be a conjunctive query,* $\sigma : Q \to \mathrm{SCTree}(\mathcal{D}, \Sigma)$ *a* $\mathrm{Consts}(\Sigma)$*-free query homomorphism, and* Z *a nonempty,* Q*-connected subset of* $\vec{z}$. *If we write* T_Z *for the set of nodes in* $\mathrm{SCTree}(\mathcal{D}, \Sigma)$ *whose bags contain a term in* $\sigma(Z)$, *then* T_Z *is a rooted subtree of* $\mathrm{SCTree}(\mathcal{D}, \Sigma)$.

Proof. For each term $t \in \sigma(Z)$, let $V_t \subseteq \mathrm{SCTree}(\mathcal{D}, \Sigma)$ be the set of nodes in which t appears. We wish to see that $T_Z = \bigcup_{t\in\sigma(Z)} V_t$ is a rooted subtree of $\mathrm{SCTree}(\mathcal{D}, \Sigma)$. Since T_Z is nonempty, it suffices to show the connectedness of T_Z.

So take two vertices $v_1, v_2 \in T_Z$, and let $z_1, z_2 \in Z$ be variables bound in Q such that $v_i \in V_{\sigma(z_i)}$ for $i \in \{1, 2\}$. As Z is Q-connected, there exists a sequence $z_1 = x_1, \ldots, x_n = z_2$ of variables in Z such that x_i and x_{i+1} are Q-adjacent for each $1 \leq i < n$.

For each i, there exists an atom A containing both x_i and x_{i+1}. As $\sigma : Q \to \mathrm{SCTree}(\mathcal{D}, \Sigma)$ is a query homomorphism, there exists a node v in $\mathrm{SCTree}(\mathcal{D}, \Sigma)$ that contains $\sigma(A)$. In particular, v contains both $\sigma(x_i)$ and $\sigma(x_{i+1})$, so $V_\sigma(x_i) \cap V_\sigma(x_{i+1}) \neq \emptyset$.

By Theorem 3.15, each $V_\sigma(x_i)$ is connected, so there is a path in $\mathrm{SCTree}(\mathcal{D}, \Sigma)$ that joins v_1 and v_2 through intersections $V_\sigma(x_i) \cap V_\sigma(x_{i+1})$. $\square$

We will use Theorem 3.17 to develop a query-answering procedure. To begin with, we define a way to split the query entailment checking problem into smaller problems by a partial guess of the query homomorphism.

Definition 3.18. Let $Q = \exists \vec{z}. \bigwedge_{j\in J} A_j$ be a conjunctive query and $\sigma_{\mathrm{commit}} : \mathrm{Vars}(Q) \rightharpoonup T$ a partial map from query variables to a set T of terms. Let $\mathrm{BVars} = \mathrm{dom}(\sigma_{\mathrm{commit}})$.

We define the *committed part* $\mathrm{Commq}(Q, \sigma_{\mathrm{commit}})$ *of* Q *according to* σ_{commit} as the variable-free query

$$\mathrm{Commq}(Q, \sigma_{\mathrm{commit}}) = \bigwedge_{\substack{j\in J \\ \mathrm{Vars}(A_j)\subseteq \mathrm{BVars}}} \sigma_{\mathrm{commit}}(A_j).$$

For each Q-connected component C of $(\vec{z} \setminus \mathrm{BVars})$, the *subquery* $\mathrm{Subq}(Q, \sigma_{\mathrm{commit}}, C)$ *of* Q *induced by* σ_{commit} *and* C is the connected Boolean conjunctive query defined by

$$\mathrm{Subq}(Q, \sigma_{\mathrm{commit}}, C) = \exists \vec{C}. \bigwedge_{j\in J'} \sigma_{\mathrm{commit}}(A_j),$$

where

$$J' = \{j \in J \mid \mathrm{Vars}(A_j) \subseteq C \cup \mathrm{BVars}\}.$$

$\square$

Example 3.19. Let

$$\begin{aligned} Q = \exists z_1, z_2, z_3, z_4, z_5, z_6.\ & S(z_1, z_2) \wedge S(z_1, z_3) \wedge R(z_2, z_3) \\ & \wedge R(z_3, z_4) \wedge R(z_3, z_5) \wedge T(z_2, z_6, z_5) \end{aligned}$$

as in Theorem 3.14. Suppose that σ_{commit} is given by

$$\begin{array}{rccc} \sigma_{\mathrm{commit}} : & \mathrm{Vars}(Q) & \rightharpoonup & \{c_1, c_2, c_3\} \\ & z_2 & \mapsto & c_1 \\ & z_3 & \mapsto & c_3 \end{array}$$

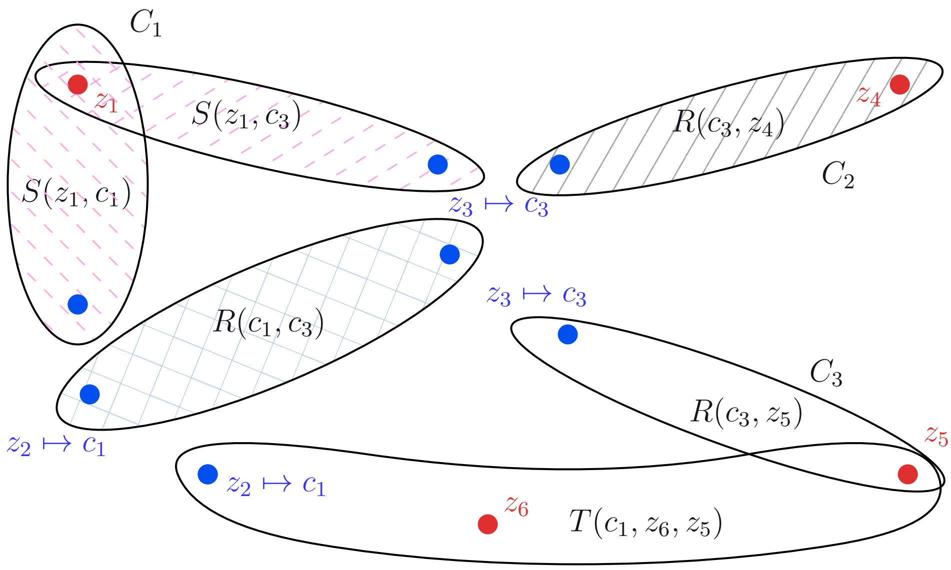

Figure 3.4: Decomposition of the query from Theorem 3.14 with partial map as in Theorem 3.19.

then $\mathrm{Commq}(Q, \sigma_{\mathrm{commit}}) = R(c_1, c_3)$. The split connected components are $C_1 = \{z_1\}$, $C_2 = \{z_4\}$ and $C_3 = \{z_5, z_6\}$, and their corresponding subqueries are

$$\begin{aligned}
\mathrm{Subq}(Q, \sigma_{\mathrm{commit}}, C_1) &= \exists z_1.\ S(z_1, c_1) \wedge S(z_1, c_3) \\
\mathrm{Subq}(Q, \sigma_{\mathrm{commit}}, C_2) &= \exists z_4.\ R(c_3, z_4) \\
\mathrm{Subq}(Q, \sigma_{\mathrm{commit}}, C_3) &= \exists z_5, z_6.\ R(c_3, z_5) \wedge T(c_1, z_6, z_5)
\end{aligned}$$

The decomposition after applying σ_{commit} is illustrated in Figure 3.4. □

Intuitively, the partial map σ_{commit} represents a partial commitment towards constructing the whole homomorphism. $\mathrm{Commq}(Q, \sigma_{\mathrm{commit}})$ is a collection of atoms which we expect to see in (the Datalog saturation of) the input instance, and for each C, $\mathrm{Subq}(Q, \sigma_{\mathrm{commit}}, C)$ is a query whose entailment tells us if σ_{commit} had been a good choice.

As the following lemma states, the original query is satisfied precisely when all split queries are.

Lemma 3.20 (Base-connected query decomposition). *Let $Q = \exists \vec{z}. \bigwedge_{j \in J} A_j$ be a Boolean conjunctive query, $\mathcal{D}$ an instance and Σ a finite set of single-headed GTGDs. Then $\mathcal{D} \cup \Sigma \models Q$ if and only if there exists a partial map $\sigma_{\mathrm{partial}} : \vec{z} \rightharpoonup \mathrm{Terms}(\mathcal{D}) \cup \mathrm{Consts}(\Sigma)$ such that*

- $\mathcal{D} \cup \Sigma \models \mathrm{Commq}(Q, \sigma_{\mathrm{partial}})$
- $\mathcal{D} \cup \Sigma \models \mathrm{Subq}(Q, \sigma_{\mathrm{partial}}, C)$ *for each Q-connected component C of* $(\vec{z} \setminus \mathrm{dom}(\sigma_{\mathrm{partial}}))$.

Proof. ($\Longrightarrow$): Suppose $\mathcal{D} \cup \Sigma \models Q$. Then by Theorem 3.11, there exists a query homomorphism $\sigma : Q \to \mathrm{Terms}(\mathrm{SCTree}(\mathcal{D}, \Sigma))$. Let

$$B = \{v \in \vec{z} \mid \sigma(v) \in \mathrm{Terms}(\mathcal{D}) \cup \mathrm{Consts}(\Sigma)\},$$

then $(\sigma \upharpoonright B)$ is a partial map satisfying conditions in the lemma.

($\Longleftarrow$): Suppose that $\sigma_{\mathrm{partial}}$ satisfies the two conditions, and let $C_1, \ldots, C_n$ be Q-connected components of $(\vec{z} \setminus \mathrm{dom}(\sigma_{\mathrm{partial}}))$. For each $1 \leq i \leq n$, we can take a query homomorphism $\sigma_{C_i} : \mathrm{Subq}(Q, \sigma_{\mathrm{partial}}, C_i) \to \mathrm{SCTree}(\mathcal{D}, \Sigma)$ by Theorem 3.11. Since all of $\sigma_{\mathrm{partial}}, \sigma_{C_1}, \ldots, \sigma_{C_n}$ have disjoint domains, $\sigma = \sigma_{\mathrm{partial}} \cup \sigma_{C_1} \cup \ldots \cup \sigma_{C_n}$ is a function $\sigma : \vec{z} \to \mathrm{Terms}(\mathrm{SCTree}(\mathcal{D}, \Sigma))$.

Since every atom A_j in Q appears either in $\mathrm{Commq}(Q, \sigma_{\mathrm{partial}})$ or in precisely one $\mathrm{Subq}(Q, \sigma_{\mathrm{partial}}, C_i)$, σ is a query homomorphism $\sigma : Q \to \mathrm{SCTree}(\mathcal{D}, \Sigma)$, and therefore $\mathcal{D} \cup \Sigma \models Q$. $\square$

We can easily extend Theorem 3.20 to non-Boolean conjunctive queries.

Corollary 3.21. *Let $Q = \exists \vec{z}. \bigwedge_{j \in J} A_j$ be a conjunctive query, $\mathcal{D}$ an instance and Σ a finite set of single-headed GTGDs.*

Then for a substitution $\alpha : \mathrm{FV}(Q) \to \mathrm{Terms}(\mathcal{D})$, $\mathcal{D} \cup \Sigma \models \alpha(Q)$ if and only if there exists an extension $\sigma_{\mathrm{partial}} : \mathrm{Vars}(Q) \rightharpoonup \mathrm{Terms}(\mathcal{D})$ of α satisfying conditions as in Theorem 3.20.

In order to produce answers to the query using Theorem 3.21, we need to decide whether or not $\mathcal{D} \cup \Sigma \models \mathrm{Subq}(Q, \sigma_{\mathrm{partial}}, C)$ holds. It turns out that we can exploit the connectedness of $\mathrm{Subq}(Q, \sigma_{\mathrm{partial}}, C)$ and the corecursive structure of $\mathrm{SCTree}(\mathcal{D}, \Sigma)$ (Theorem 3.12) to check the query entailment recursively. To make this precise, we define the notion of a *successful commit point*, a point at which a connected Boolean conjunctive query can be split in a way similar to Theorem 3.20.

Definition 3.22. Let Q be a connected Boolean conjunctive query and $\mathrm{SCTree}(\mathcal{D}, \Sigma)$ a shortcutting chase tree. We say that a vertex $v \in \mathrm{SCTree}(\mathcal{D}, \Sigma)$ with the associated bag $\mu(v)$ of facts is a *successful commit point for Q in* $\mathrm{SCTree}(\mathcal{D}, \Sigma)$ if there exists a $\mathrm{Consts}(\Sigma)$-free partial map $\sigma_{\mathrm{commit}} : \mathrm{Vars}(Q) \rightharpoonup \mathrm{Terms}(\mu(v)) \setminus \mathrm{Consts}(\Sigma)$, which we call a *commit map at v*, such that

- $\mathrm{dom}(\sigma_{\mathrm{commit}})$ is nonempty,
- $\mu(v) \models \mathrm{Commq}(Q, \sigma_{\mathrm{commit}})$, and
- for each Q-connected component C of $(\mathrm{Vars}(Q) \setminus \mathrm{dom}(\sigma_{\mathrm{commit}}))$, there exists a $\mathrm{Consts}(\Sigma)$-free query homomorphism $\sigma_C : \mathrm{Subq}(Q, \sigma_{\mathrm{commit}}, C) \to \mathrm{SCTree}(\mu(v), \Sigma)$.

$\square$

To witness the entailment of a connected Boolean conjunctive query (with all query variables that get mapped to $\mathrm{Consts}(\Sigma)$ already substituted), finding a *single* successful commit point in the shortcutting chase tree suffices.

Theorem 3.23. *(Recursive BCQ Entailment) Let $Q = \exists \vec{z}. \bigwedge_{j \in J} A_j$ be a Boolean conjunctive query, $\mathcal{D}$ an instance and Σ a finite set of single-headed GTGDs. Then there exists a* $\mathrm{Consts}(\Sigma)$*-free query homomorphism $\sigma : Q \to \mathrm{SCTree}(\mathcal{D}, \Sigma)$ if and only if there exists a successful commit point for Q in* $\mathrm{SCTree}(\mathcal{D}, \Sigma)$*.*

Proof. ($\Longrightarrow$): Suppose that $\sigma : Q \to \mathrm{SCTree}(\mathcal{D}, \Sigma)$ is a $\mathrm{Consts}(\Sigma)$-free query homomorphism. By Theorem 3.17, the set $T_{\mathrm{Vars}(Q)}$ of nodes in which terms in $\sigma(\mathrm{Vars}(Q))$ appear is a rooted tree in $\mathrm{SCTree}(\mathcal{D}, \Sigma)$. Let r be the root of $T_{\mathrm{Vars}(Q)}$ and $\mu(r)$ the bag of facts at r in $\mathrm{SCTree}(\mathcal{D}, \Sigma)$. We aim to show that r is a successful commit point for Q.

By Theorem 3.12, $T_{\mathrm{Vars}(Q)} = \mathrm{SCTree}(\mu(r), \Sigma)$. Since terms in $\sigma(\mathrm{Vars}(Q))$ only appear in $T_{\mathrm{Vars}(Q)}$, σ is a $\mathrm{Consts}(\Sigma)$-free query homomorphism $\sigma : Q \to \mathrm{SCTree}(\mu(r), \Sigma)$. Let $B = \{v \in \mathrm{Vars}(Q) \mid \sigma(v) \in \mathrm{Terms}(\mu(r))\}$ be a nonempty set of variables mapped to terms in r and define $\sigma_{\mathrm{commit}} = \sigma \restriction B$, then σ_{commit} is a commit map at r.

($\Longleftarrow$): Suppose that v is a successful commit point with a commit map σ_{commit}. Let $C_1, \ldots, C_n$ be Q-connected components of $(\mathrm{Vars}(Q) \setminus \mathrm{dom}(\sigma_{\mathrm{commit}}))$. Then for each $1 \leq i \leq n$, there is a $\mathrm{Consts}(\Sigma)$-free query homomorphism σ_{C_i} : $\mathrm{Subq}(Q, \sigma_{\mathrm{commit}}, C_i) \to \mathrm{SCTree}(\mu(v), \Sigma)$. If we let $\sigma = \sigma_{\mathrm{commit}} \cup \sigma_{C_1} \cup \ldots \cup \sigma_{C_n}$, then we can check that $\sigma : Q \to \mathrm{SCTree}(\mathcal{D}, \Sigma)$ is a $\mathrm{Consts}(\Sigma)$-free query homomorphism as in the proof of Theorem 3.20. □

Remark 3.24. Notice that, to decide whether a vertex v in $\mathrm{SCTree}(\mathcal{D}, \Sigma)$ is a successful commit point for Q with a commit map σ_{commit}, we also need to decide if subqueries $\mathrm{Subq}(Q, \sigma_{\mathrm{commit}}, C_i)$ are satisfied in the subtree $\mathrm{SCTree}(\mu(v), \Sigma)$. As we require the commit map σ_{commit} to be a nonempty map, the number of bound variables in each subquery $\mathrm{Subq}(Q, \sigma_{\mathrm{commit}}, C_i)$ is strictly less than that of Q.

Moreover, for Boolean conjunctive queries, the query entailment is unaffected by a certain renaming of constants in the root instance. Therefore, we only need to search for successful commit points up to renaming equivalence. □

We capture the latter intuition with the following.

Definition 3.25. Let $\mathcal{D}$ be an instance. A *Σ-preserving renaming on $\mathcal{D}$* is an injective function $\sigma : \mathrm{Terms}(\mathcal{D}) \setminus \mathrm{Consts}(\Sigma) \hookrightarrow T$ where T is a set of terms with $T \cap \mathrm{Consts}(\Sigma) = \emptyset$.

Proposition 3.26. *If Q is a Boolean conjunctive query and σ is Σ-preserving renaming on an instance $\mathcal{D}$, then $\mathcal{D} \cup \Sigma \models Q$ if and only if $\sigma(\mathcal{D}) \cup \Sigma \models Q$.*

Proof. If we extend σ to all of $\mathrm{Terms}(\mathrm{SCTree}(\mathcal{D}, \Sigma))$ by defining

$$\sigma'(t) = \begin{cases} \sigma(t) & \text{if } t \in \mathrm{Consts}(\mathcal{D}) \setminus \mathrm{Consts}(\Sigma) \\ t & \text{otherwise} \end{cases}$$

Then $\mathrm{SCTree}(\sigma'(\mathcal{D}), \Sigma)$ is an isomorphic image of $\mathrm{SCTree}(\mathcal{D}, \Sigma)$ under σ'. Now apply Theorem 3.11. □

Remark 3.27. In Theorem 3.26, we require that the renaming σ *preserves* all constants in Σ. This is because

- if $c \in \mathrm{Consts}(\Sigma)$ is renamed to some other constant by σ, then a rule in Σ that fired in $\mathrm{SCTree}(\mathcal{D}, \Sigma)$ may no longer fire in $\mathrm{SCTree}(\sigma(\mathcal{D}), \Sigma)$ thereby invalidating the ($\Longrightarrow$) implication, and

- if $t \in \mathrm{Terms}(\mathcal{D})$ is renamed to a constant in Σ, then a rule that did not fire in $\mathrm{SCTree}(\mathcal{D}, \Sigma)$ may fire in $\mathrm{SCTree}(\sigma(\mathcal{D}), \Sigma)$, invalidating ($\Longleftarrow$) direction.

Definition 3.28. We say that two instances $\mathcal{D}_1, \mathcal{D}_2$ are *Σ-renaming-equivalent* (written $\mathcal{D}_1 \cong_\Sigma \mathcal{D}_2$) if there exists a Σ-preserving renaming σ with $\sigma(\mathcal{D}_1) = \mathcal{D}_2$. It is easy to see that $\cong_\Sigma$ is an equivalence relation.

Proposition 3.29. *For any shortcutting chase tree* $\mathrm{SCTree}(\mathcal{D}, \Sigma)$*, there are only finitely many $\cong_\Sigma$-equivalence classes of bags in* $\mathrm{SCTree}(\mathcal{D}, \Sigma)$*.*

Proof. For any non-root node v of $\mathrm{SCTree}(\mathcal{D}, \Sigma)$, by Theorem 3.9 there are at most $K + |\mathrm{Consts}(\Sigma)|$ terms present in the bag $\mu(v)$. Hence there are at most $|P| \cdot 2^{(K+|\mathrm{Consts}(\Sigma)|)}$ facts in $\mu(v)$, where P is the set of predicates in $\mathcal{D} \cup \Sigma$. In particular, the number of $\cong_\Sigma$-equivalence classes in $\mathrm{SCTree}(\mathcal{D}, \Sigma)$ is at most $2^{|P| \cdot 2^{(K+|\mathrm{Consts}(\Sigma)|)}} + 1$, where the bag at the root node accounts for $(+1)$. $\square$

We are ready to present a query-answering procedure Algorithm 2 based on the intuition of Theorem 3.24. Note that, in EntailsConnectedBCQ, we can scan through all equivalence classes $[\mathcal{D}]_{\cong_\Sigma}$ of bags in $\mathrm{SCTree}(\mathcal{D}, \Sigma)$ by a depth-first search, since

- there are only finitely many such classes by Theorem 3.29, and

- for each such $\cong_\Sigma$-equivalence class $\mathcal{E}$, there exists a path $v_1, \ldots, v_n$ from the root node of $\mathrm{SCTree}(\mathcal{D}, \Sigma)$ such that

  1. none of $\mu(v_1), \ldots, \mu(v_n)$ are $\cong_\Sigma$-equivalent
  2. $[\mu(v_n)]_{\cong_\Sigma} = \mathcal{E}$

  because we can always shorten a path not satisfying (1) by replacing a segment $v_i, \ldots, v_{i+m}$ such that $\mu(v_i) \cong_\Sigma \mu(v_{i+m})$ with just v_i, and then replacing all of $v_{i+m+1}, \ldots, v_n$ by their $\cong_\Sigma$-equivalent copies in the subtree rooted at v_i.

We will only use Algorithm 2 as an intermediate step towards producing a rewriting. We will not go through the algorithm line by line, but we shall sketch the proof of its correctness and termination.

Theorem 3.30. *The procedure* AnswerConjunctiveQuery *in Algorithm 2 produces all valid answers to GTGDs-CQ Answering in finite time.*

Proof (sketch). Termination is clear since

- all **for**-loops iterate over a finite set, and
- size of connected BCQ decreases on every recursive call (Theorem 3.24).

We can first prove the correctness of SatisfiedWith for all connected BCQs by induction on the number of bound variables and recursively applying Theorem 3.23. It is then straightforward to see, by applying Theorem 3.21, that AnswerConjunctiveQuery produces all valid answers. □

Algorithm 2 Basic query answering procedure

```
// decide if there exists a Consts(Σ)-free query homomorphism
// σ : Q → SCTree(𝒟, Σ) for connected BCQ Q
procedure EntailsConnectedBCQ(𝒟, Σ, connected BCQ Q)
    for each equivalence class [𝒢]_{≅_Σ} of bags in SCTree(𝒟, Σ) do
        if IsSuccessfulCommitPoint(𝒢, Σ, Q) then
            return true
        end if
    end for
    return false
end procedure

procedure IsSuccessfulCommitPoint(𝒟, Σ, connected BCQ Q)
    for each nonempty σ_commit : Vars(Q) ⇀ Terms(𝒟) \ Consts(Σ) do
        if SatisfiedWith(𝒟, Σ, Q, σ_commit) then
            return true
        end if
    end for
    return false
end procedure

// computes if the query is satisfied with a partial substitution σ_partial
procedure SatisfiedWith(𝒟, Σ, Q, σ_partial : Vars(Q) ⇀ Terms(𝒟))
    require dom(σ_partial) ⊇ FV(Q)
    𝒟_Dsat ← DatalogSaturate(ARew(Σ), 𝒟)
    baseSatisfied ← 𝒟_Dsat ⊨ Commq(Q, σ_partial)
    C_1, ..., C_n ← Q-connected components of Vars(Q) \ dom(σ_partial)
    allComponentsSatisfied ← ⋀_{i=1}^{n}
        EntailsConnectedBCQ(𝒟_Dsat, Σ, Subq(Q, σ_partial, C_i))
    return baseSatisfied and allComponentsSatisfied
end procedure

procedure AnswerConjunctiveQuery(𝒟, Σ, conjunctive query Q)
    for each substitution α : FV(Q) → Terms(𝒟) ∪ Consts(Σ) do
        for each substitution σ_Consts(Σ) : (Vars(Q) \ FV(Q)) ⇀ Consts(Σ) do
            if SatisfiedWith(𝒟, Σ, Q, α ∪ σ_Consts(Σ)) then
                output α as an answer
                break inner loop
            end if
        end for
    end for
end procedure
```

Chapter 4

Deriving a Rewriting

In Chapter 3, we developed Algorithm 2 for producing all answers to a conjunctive query over single-headed GTGDs.

However, because of high data complexity, the algorithm is impractical for query-answering purposes: We have to explore a part of $\mathrm{SCTree}(\mathcal{D}, \Sigma)$ for every single substitution $\alpha : \mathrm{FV}(Q) \to \mathrm{Consts}(\mathcal{D})$. So instead, we aim to use Algorithm 2 as a stepping stone to devising a Datalog rewriting that works for arbitrary input $\mathcal{D}$.

The first important observation is that the entailment of a subquery in a proper subtree of $\mathrm{SCTree}(\mathcal{D}, \Sigma)$ only depends on a fraction of $\mathcal{D}$.

More precisely, suppose that a connected subquery Q_{sub} is satisfied in a subtree T_c of $\mathrm{SCTree}(\mathcal{D}, \Sigma)$ rooted at a node c of the root node r so that there is a query homomorphism $h : Q_{\mathrm{sub}} \to T_c$. Suppose that c is obtained from r by firing an existential rule $\tau = \forall \vec{x}.\ \beta \to \exists \vec{y}.\ H$ together with a substitution $\sigma : \vec{x} \to \mathrm{Terms}(\mathcal{D})$. If we let $\mathcal{D}'$ be a subset of $\mathcal{D}$ formed by extracting facts in $\mathcal{D}$ that are either

- used in the substituted body $\sigma(\beta)$, or
- inherited by c, i.e. facts whose arguments are all in $\sigma(H)$,

then we can still fire τ with σ in $\mathrm{SCTree}(\mathcal{D}', \Sigma)$ to obtain a node c'. As c' inherits the same set of facts, the tree T_c and the subtree $T_{c'}$ rooted at c' are isomorphic, and in particular, h restricts to $Q_{\mathrm{sub}} \to T_{c'}$. As $\mathcal{D}'$ is guarded by $\sigma(\beta)$, $\mathcal{D}'$ is (K, Σ)-small. Ultimately, if we write

$$K = \max_{P \in \mathrm{Predicates}(\Sigma \cup \{Q\})} \mathrm{Arity}(P)\,,$$

we only need a (K, Σ)-small set of facts to entail a subquery in a proper subtree.

Moreover, by Theorem 3.26, we are only interested in the *structure* of the (K, Σ)-small instance that is needed to satisfy a subquery. For example, suppose that we have found out that an instance $\{R(c_1, c_2), R(c_2, r_1)\}$ is sufficient to entail a subquery $\exists z.\ T(c_2, r_2, z)$, where only r_1 and r_2 are constants in Σ. We can replace c_1 and c_2 with any other constants and still obtain a valid implication such as $R(c_4, c_6) \wedge R(c_6, r_1) \to \exists z.\ T(c_6, r_2, z)$. By a generalisation rule, we can deduce that $\forall x_1, x_2.\ R(x_1, x_2) \wedge R(x_2, r_1) \to \exists z.\ T(x_2, r_2, z)$ under Σ.

With these observations in mind, we adopt the following strategy for building a rewriting.

- Start with an atomic rewriting $\mathrm{A}\textsc{Rew}(\Sigma)$ of Σ.
- For each (*not* necessarily maximal) Q-connected nonempty set C of bound variables, introduce an intensional predicate $\mathrm{Subgoal}_C$, which asserts that a subquery induced by C is satisfied by some partial substitution on Q.
- For each *local instance* I (to be defined in Section 4.1), which represents a (K, Σ)-small structure in the base instance, decide if I contains enough facts to entail a subquery induced by C. If so, add a rule roughly of the form $I \to \mathrm{Subgoal}_C$.
- Finally, add all rules that *integrate* subgoals into the goal atom. These rules essentially perform the inverse of Theorem 3.21 by gathering base facts and subgoal atoms together to infer the existence of a query homomorphism.

In Section 4.1, we define the precise structure of *local instances* and what it means for one to entail a subquery. We then briefly describe how to enumerate instances that entail a subquery in Section 4.2, putting pieces together in Section 4.3 to present a rewriting algorithm. Finally, we discuss in Section 4.4 some optimisations on the naive algorithm presented in Section 4.2.

4.1 The Subquery Entailment Problem

To proceed with the strategy, we need to define a data structure that can be input to the following problem.

Problem 4.1 (Informal)**.** Suppose I is a certain (K, Σ)-small structure, C a Q-connected set of bound variables and $Q_{\mathrm{sub},C}$ a subquery of Q with existentials C. Does I contain enough facts to entail $Q_{\mathrm{sub},C}$?

4.1.1 Local Instances

First, we formalise the "(K, Σ)-small structure".

Since there are at most K (maximum arity of predicates in $\mathcal{D} \cup \Sigma$) non-$\mathrm{Consts}(\Sigma)$-terms in a (K, Σ)-small bag of facts, we might consider relabelling them with numbers $\{1, 2, \ldots, K\}$. However, if we completely identify bags with $\cong_\Sigma$-equivalence, we will no longer be able to recover the structure of $\mathrm{SCTree}(\mathcal{D}, \Sigma)$. Figure 4.1 illustrates the issue.

A trick to solve this issue is to use $2K$ labels, which we call *local names*, with the convention that the same label in adjacent bags represents a term shared between them, as illustrated in Figure 4.2. Formally, we work with the following structure.

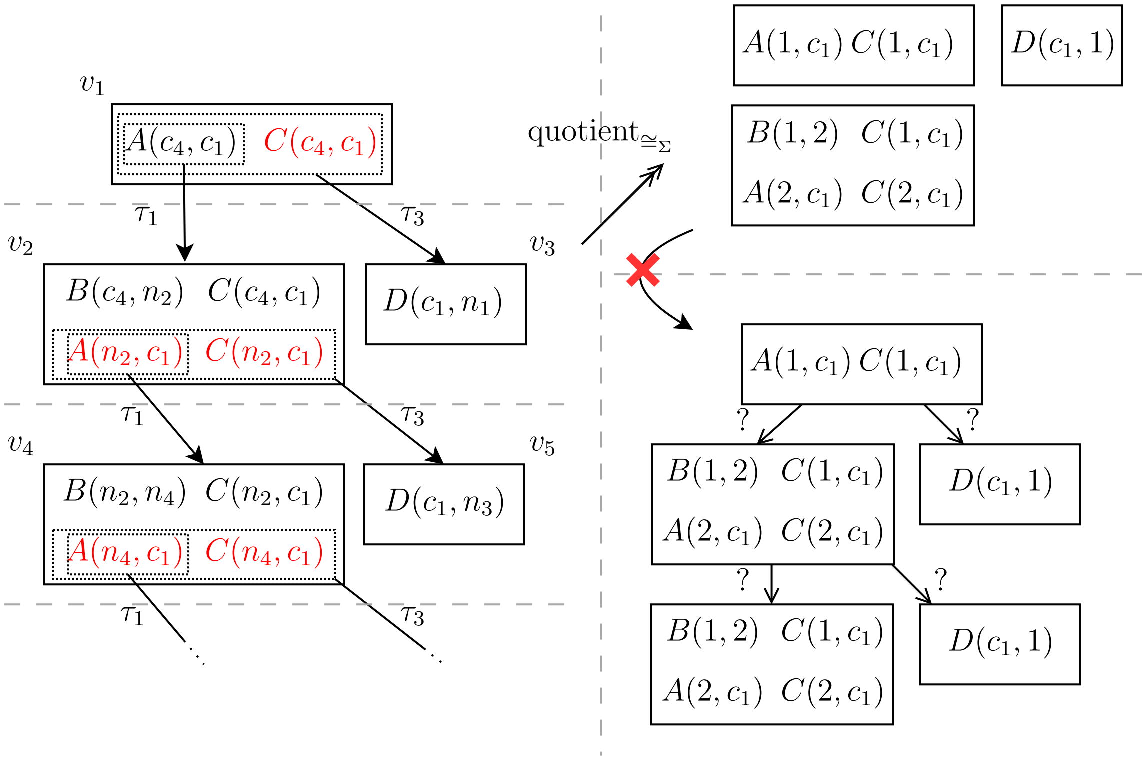

Figure 4.1: (Left) The shortcutting chase from Figure 3.2. (Top right) If we identify bags with $\cong_\Sigma$ by relabelling terms with numbers, we get three bags representing $\cong_\Sigma$-equivalence classes. (Bottom right) We can no longer assemble these equivalence classes back to a chase structure since there is no way to distinguish between the inheritance and the introduction of terms in a chased node.

Definition 4.2. A (K, Σ)-*local instance* is a bag I of facts such that

- for every fact $F \in I$, every term in F is either
  - a constant in Σ, or
  - a *local name* from the set $\{1, \ldots, 2K\}$ of fresh constants
- there are at most K local names that *are active* (i.e. appear) in I.

We write LNames(I) for the set of active local names in I. □

Notice that, as shown in Figure 4.2, we can translate a SCTree$(\mathcal{D}, \Sigma)$ into a tree-like structure that uses local names. By abuse of notation, we write SCTree(I, Σ) for the tree-like structure obtained this way.

The structure SCTree(I, Σ) can be constructed in a way similar to how we construct an ordinary shortcutting chase tree, except that when firing an existential rule, we *reuse* a local name that was inactive at the parent node instead of introducing fresh nulls. For example, at v_4' in Figure 4.2, we are using a local name 3, which is inactive at the parent node v_2', in place of the null n_4. We no longer include the local name 1 in v_4' because v_4' inherits no fact containing 1.

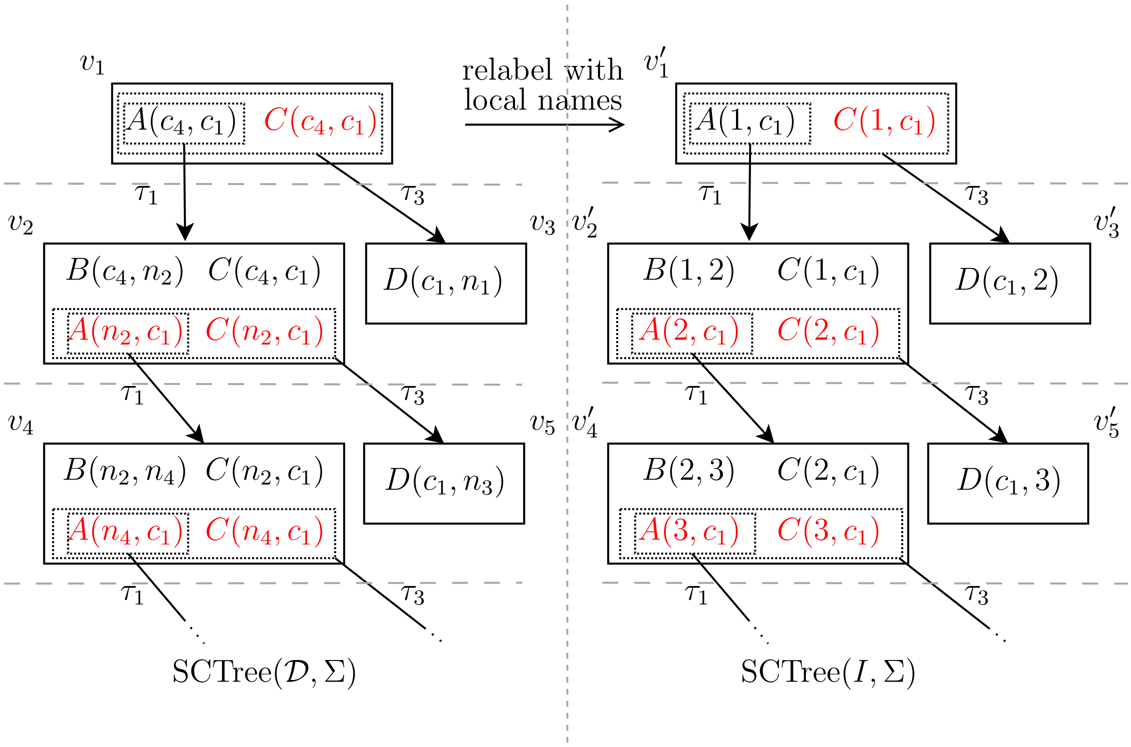

Figure 4.2: (Left) The shortcutting chase from Figure 3.2. (Right) relabelling of the shortcutting chase tree with local names. Two equal local names in sibling nodes (e.g. 2 in v'_2 and v'_3) may represent different terms, while they represent the same term in parent-child nodes (e.g. 2 in v'_2 and v'_4).

We say that such local names are *dropped* by the chase step. If a local name n dropped at v becomes active again in a descendant v' of v, then n at v and v' represent different terms.

4.1.2 Partially Substituted Subqueries

Our next task is to represent a partially substituted subquery. For a Q-connected set C of variables, the subquery induced by Q should

- only contain atoms in Q that mention a variable from C, and
- have all variables not in C substituted with a term.

It is convenient to introduce the following notion to describe the latter condition.

Definition 4.3. For a conjunctive query $Q = \exists \vec{z}. \bigwedge_{j \in J} A_j$ and a Q-connected subset C of $\vec{z}$, the Q-*boundary* (written $\partial_Q C$) of C is the set of (either free or bound) variables in Q defined by

$$\partial_Q C = \left\{ v \in \mathrm{Vars}(Q) \setminus C \;\middle|\; \begin{array}{l} \exists j \in J \text{ such that} \\ \mathrm{Vars}(A_j) \cap C \neq \emptyset \text{ and } v \in \mathrm{Vars}(A_j) \end{array} \right\}$$

□

Example 4.4. Let

$$Q = \exists z_1, z_2, z_3, z_4, z_5, z_6.\ S(z_1, z_2) \wedge S(z_1, z_3) \wedge R(z_2, z_3) \\ \wedge R(z_3, z_4) \wedge R(z_3, z_5) \wedge T(z_2, z_6, z_5)$$

as in Theorem 3.14. Then $\partial_Q\{z_4\} = \{z_3\}$, $\partial_Q\{z_2\} = \{z_1, z_3, z_5, z_6\}$ and $\partial_Q\{z_5, z_6\} = \{z_2, z_3\}$. We illustrate the first example in Figure 4.3. $\square$

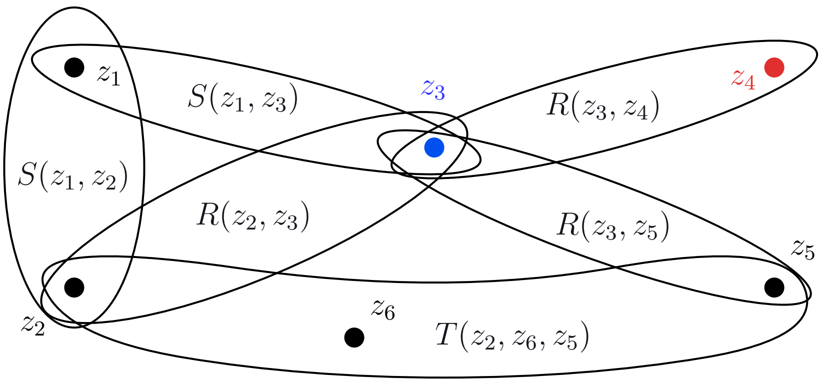

Figure 4.3: For the query from Theorem 3.14, the Q-boundary $\partial_Q\{z_4\}$ of $\{z_4\}$ (red) is $\{z_3\}$ (blue).

During a search for a successful commit point in $\mathrm{SCTree}(I, \Sigma)$, a variable in $\partial_Q C$ is mapped either

- to a constant in $\mathrm{Consts}(\Sigma)$, or
- to a local name in I.

With this observation, we are almost ready to define the notion of a partially substituted subquery. However, before we proceed, we remark on the following corner case regarding constants in the query.

Assume for a moment that there are no rule constants. Observe that, for a local instance I to entail a subquery $\exists y.\ T(1, y, c)$ containing a constant c, there must be some fact in I that contains c since otherwise c is never introduced to the shortcutting chase tree.

This remark motivates the following definition.

Definition 4.5. Let $Q = \exists \vec{z}.\ \bigwedge_{j\in J} A_j$ be a conjunctive query, C be a Q-connected subset of variables bound in Q and I a (K, Σ)-local instance. An *embedding of constants adjacent to C into I* is an injective map $\iota : S_{Q,C} \hookrightarrow \mathrm{LNames}(I)$, where $S_{Q,C}$ is the set of constants in the subquery, defined by

$$S_{Q,C} = \left\{ c \in \mathrm{Consts}(Q) \setminus \mathrm{Consts}(\Sigma) \;\middle|\; \begin{array}{l} \exists j \in J \text{ such that} \\ \mathrm{Vars}(A_j) \cap C \neq \emptyset \text{ and } c \in \mathrm{Consts}(A_j) \end{array} \right\}.$$

Putting these concepts together, we obtain the following notion of a subquery.

Definition 4.6. Let Q be a conjunctive query, C be a Q-connected subset of variables bound in Q and I a (K, Σ)-local instance. A *representation of a subquery of Q induced by C and local to I* is a triple $(\sigma_{\mathrm{Consts}(\Sigma)}, \sigma_I, \iota)$ of mappings

$$\begin{array}{rcll} \sigma_{\mathrm{Consts}(\Sigma)}: & \partial_Q C & \rightharpoonup & \mathrm{Consts}(\Sigma) \\ \sigma_I: & \partial_Q C & \rightharpoonup & \mathrm{LNames}(I) \\ \iota: & S_{Q,C} & \hookrightarrow & \mathrm{LNames}(I) \end{array}$$

such that

- $\{\mathrm{dom}\left(\sigma_{\mathrm{Consts}(\Sigma)}\right), \mathrm{dom}(\sigma_I)\}$ is a partition of $\partial_Q C$, and
- ι is an embedding of constants adjacent to C into I.

Given such a triple, its *local realisation at I*, written $\mathrm{LRealz}_Q\left(\sigma_{\mathrm{Consts}(\Sigma)}, \sigma_I, \iota\right)$, is the connected Boolean conjunctive query

$$\mathrm{LRealz}_Q\left(\sigma_{\mathrm{Consts}(\Sigma)}, \sigma_I, \iota\right) = \exists \vec{C}.\ \bigwedge_{j \in J_C} (\sigma_{\mathrm{Consts}(\Sigma)} \cup \sigma_I \cup \iota)(A_j)$$

mentioning local names, where $J_C = \{j \in J \mid \mathrm{Vars}(A_j) \cap C \neq \emptyset\}$.

□

Example 4.7. Let

$$Q = \exists z_1, z_2, z_3, z_4.\ S(c_1, z_1, z_4, z_2) \wedge T(z_1, c_2, z_3) \wedge T(z_3, c_1, z_4),$$

where c_1, c_2 are constants not in Σ, $C = \{z_1, z_2\}$ and

$$I = \{S(1, 2, 3, 4), T(4, 2, 1), R(2, 3)\}.$$

Then $\partial_Q C = \{z_3, z_4\}$ and $S_{Q,C} = \{c_1, c_2\}$. If we let

$$\begin{array}{rcl} \sigma_{\mathrm{Consts}(\Sigma)} & = & \{z_3 \mapsto r_1\} \\ \sigma_I & = & \{z_4 \mapsto 2\} \\ \iota & = & \{c_1 \mapsto 1, c_2 \mapsto 3\} \end{array}$$

where $r_1 \in \mathrm{Consts}(\Sigma)$, then

$$\mathrm{LRealz}_Q\left(\sigma_{\mathrm{Consts}(\Sigma)}, \sigma_I, \iota\right) = \exists z_1, z_2.\ S(1, z_1, 2, z_2) \wedge T(z_1, 3, r_1).$$

□

4.1.3 Formalising the Subquery Entailment Problem

Combining notions defined in Section 4.1.1 and Section 4.1.2, we can formally re-define the Theorem 4.1.

Definition 4.8. Let $Q = \exists \vec{z}. \bigwedge_{j \in J} A_j$ be a conjunctive query. A (Σ, Q)-*subquery entailment problem instance* is a 5-tuple $(C, I, \sigma_{\mathrm{Consts}(\Sigma)}, \sigma_I, \iota)$ where

- C is a Q-connected set of variables bound in Q,
- I is a (K, Σ)-local instance, where

$$K = \max_{P \in \mathrm{Predicates}(\Sigma \cup \{Q\})} \mathrm{Arity}(P)$$

, and

- $(\sigma_{\mathrm{Consts}(\Sigma)}, \sigma_I, \iota)$ is a representation of a subquery of Q induced by C and local to I.

$\square$

Problem 4.9 (Subquery Entailment Problem, formalisation of Theorem 4.1)**.** Given a (Σ, Q)-subquery entailment problem instance $\mathcal{I} = (C, I, \sigma_{\mathrm{Consts}(\Sigma)}, \sigma_I, \iota)$, does $I \cup \Sigma \models \mathrm{LRealz}_Q\big(\sigma_{\mathrm{Consts}(\Sigma)}, \sigma_I, \iota\big)$ hold? $\square$

We call an instance $\mathcal{I}$ a *subquery entailment* if it is an YES instance of Theorem 4.9

4.2 The Naive Subquery Entailment Enumeration

We can solve Theorem 4.9 with only a minor modification to Algorithm 2, with the basic idea being that we descend the $\mathrm{SCTree}(I, \Sigma)$ to search for a successful commit point. The only difference is that when we fire an existential rule, we must not drop local names that appear in the substituted query Q_{subst}, since otherwise Q_{subst} becomes immediately unsatisfiable in the chased subtree. The first version of our algorithm (Algorithm 3) is, therefore, a verbatim translation of Algorithm 2 into the context of local instances.

Later in Section 4.4, we discuss optimisations of Algorithm 3. Nevertheless, for now, we move on to the other components of the rewriting algorithm to see how outputs from Algorithm 3 can be assembled into a Datalog rewriting.

4.3 A Rewriting Algorithm

4.3.1 From a Subquery Entailment to a Datalog Rule

Now that we can enumerate subquery entailments, we show how to turn them into Datalog rules that derive subgoals.

Suppose that C is a Q-connected set of bound variables in Q, and let Q_{sub} be the (unsubstituted) subquery induced by C. The way in which Q_{sub} is satisfied can be fully described by how variables in $\partial_Q C$ are mapped to terms. Therefore, to represent a subquery satisfaction, we introduce the *subgoal predicate* as an intensional predicate $\mathrm{Subgoal}_{Q,C}$ having the arity $|\partial_Q C|$.

Algorithm 3 Naive Subquery Entailment Enumeration

// decide if there exists a $\mathrm{Consts}(\Sigma)$-free query homomorphism
// $\sigma : Q \to \mathrm{SCTree}(I, \Sigma)$ for a BCQ Q with only
// $\mathrm{LNames}(I)$ as query constants
procedure EntailsConnectedBCQ(I, Σ, connected BCQ Q)
  for each local instance I' in $\mathrm{SCTree}(I, \Sigma)$ that can be reached without dropping local names $\mathrm{LNames}(I) \cap \mathrm{Consts}(Q)$ **do**
    if IsSuccessfulCommitPoint(I, Σ, Q) **then**
      return true
    end if
  end for
  return false
end procedure

procedure IsSuccessfulCommitPoint(I, Σ, connected BCQ Q)
  for each nonempty $\sigma_{\mathrm{commit}} : \mathrm{Vars}(Q) \rightharpoonup \mathrm{LNames}(I)$ **do**
    if BooleanQSatisfiedWith(I, Σ, Q, σ_{commit}) **then**
      return true
    end if
  end for
  return false
end procedure

// computes if Boolean Q that only has $\mathrm{LNames}(I)$ as query constants
// is satisfied with a partial substitution $\sigma_{\mathrm{partial}} : \mathrm{Vars}(Q) \rightharpoonup \mathrm{LNames}(I)$
procedure BCQSatisfiedWith(I, Σ, Q, $\sigma_{\mathrm{partial}}$)
  $I_{\mathrm{Dsat}} \leftarrow$ DatalogSaturate(ARew(Σ), I)
  baseSatisfied $\leftarrow I_{\mathrm{Dsat}} \models \mathrm{Commq}(Q, \sigma_{\mathrm{partial}})$
  $C_1, \ldots, C_n \leftarrow Q$-connected components of $\mathrm{Vars}(Q) \setminus \mathrm{dom}(\sigma_{\mathrm{partial}})$
  allComponentsSatisfied $\leftarrow \bigwedge_{i=1}^{n}$
    EntailsConnectedBCQ(I_{Dsat}, Σ, $\mathrm{Subq}(Q, \sigma_{\mathrm{partial}}, C_i)$)
  return baseSatisfied **and** allComponentsSatisfied
end procedure

procedure EnumerateSubqueryEntailments(finite set Σ of single-headed GTGDs, conjunctive query Q)
  for each (Σ, Q)-subquery entailment problem instance $\mathcal{I}$ **do**
    $(C, I, \sigma_{\mathrm{Consts}(\Sigma)}, \sigma_I, \iota) \leftarrow \mathcal{I}$
    if EntailsConnectedBCQ(I, Σ, $\mathrm{LRealz}_Q(\sigma_{\mathrm{Consts}(\Sigma)}, \sigma_I, \iota)$) **then**
      output $\mathcal{I}$
    end if
  end for
end procedure

For each subquery entailment $\mathcal{I}$, we wish to add a Datalog rule (which we will write $\mathrm{SubgoalRule}(\mathcal{I})$) that has the local instance in the body and the subgoal atom as the head. For example, suppose that the subquery in Theorem 4.7 is entailed by the local instance I given in the example. We would then like to add a rule

$$\forall x_2, x_4.\ S(c_1, x_2, c_2, x_4) \wedge T(x_4, x_2, c_1) \wedge R(x_2, c_2) \rightarrow \mathrm{Subgoal}_{Q,\{z_1,z_2\}}(r_1, x_2)$$

where x_2 is the variable replacing z_4 in the original query.

Formally, we define $\mathrm{SubgoalRule}(\mathcal{I})$ as follows.

Definition 4.10. Let $\mathcal{I} = (C, I, \sigma_{\mathrm{Consts}(\Sigma)}, \sigma_I, \iota)$ be a (Σ, Q)-subquery entailment problem instance. Let S be the subgoal atom substituted with local names, defined by

$$S = (\sigma_{\mathrm{Consts}(\Sigma)} \cup \sigma_I)\left(\mathrm{Subgoal}_{Q,C}\left(\overrightarrow{\partial_Q C}\right)\right).$$

Let ρ be a renaming of all local names to variables and query constants, defined by

$$\rho(n) = \begin{cases} \iota^{-1}(n) & \text{if } n \in \mathrm{range}(\iota) \\ x_n & \text{otherwise} \end{cases}$$

Finally, let $\mathrm{SubgoalRule}(\mathcal{I})$ be the Datalog rule given by

$$\mathrm{SubgoalRule}(\mathcal{I}) = \rho\,(I \rightarrow S)$$

with all free variables universally quantified. □

Under the identification of the subgoal atom $\mathrm{Subgoal}_{Q,C}(\vec{t})$ with the subquery $\mathrm{Subq}\left(Q, \left\{\overrightarrow{\partial_Q C} \mapsto \vec{t}\right\}, C\right)$ of Q, the rule $\mathrm{SubgoalRule}(\mathcal{I})$ is "sound" if and only if $\mathcal{I}$ is a subquery entailment. To be formal, we claim the following.

Proposition 4.11. *Let Q be a conjunctive query and $\mathcal{I} = (C, I, \sigma_{\mathrm{Consts}(\Sigma)}, \sigma_I, \iota)$ a (Σ, Q)-subquery entailment problem instance. Let $\Phi_{Q,C}$ be the formula stating the identification of subgoal facts with subquery satisfactions, given by*

$$\Phi_{Q,C} = \forall \vec{t}.\ \left(\mathrm{Subgoal}_{Q,C}(\vec{t}) \leftrightarrow \mathrm{Subq}\left(Q, \left\{\overrightarrow{\partial_Q C} \mapsto \vec{t}\right\}, C\right)\right).$$

Then $\Sigma \cup \{\Phi_{Q,C}\} \models \mathrm{SubgoalRule}(\mathcal{I})$ if and only if $\mathcal{I}$ is a subquery entailment.

Proof.

A routine calculation proves both directions.

($\Longrightarrow$): Let ρ, I, and S be as defined in Theorem 4.10. Let λ be an assignment that maps each variable x_n in $\mathrm{SubgoalRule}(\mathcal{I})$ to a local name n. Then by applying the substitution λ, we have $\Sigma \cup \{\Phi_{Q,C}\} \models \iota^{-1}(I \rightarrow S)$. Since S does not contain a local name,

$$\Sigma \cup \{\Phi_{Q,C}\} \models \iota^{-1}(I) \rightarrow (\sigma_{\mathrm{Consts}(\Sigma)} \cup \sigma_I)\left(\mathrm{Subgoal}_{Q,C}\left(\overrightarrow{\partial_Q C}\right)\right).$$

By using ($\to$) of $\Phi_{Q,C}$, we have

$$\Sigma \cup \{\Phi_{Q,C}\} \models \iota^{-1}(I) \to \mathrm{Subq}\big(Q, \sigma_{\mathrm{Consts}(\Sigma)} \cup \sigma_I, C\big)\,.$$

Since none of range(ι^{-1}) appears in $\Sigma \cup \{\Phi_{Q,C}\}$, we can generalise query constants on the right-hand side to universally quantified variables and immediately instantiate them back with original local names. Therefore

$$\Sigma \cup \{\Phi_{Q,C}\} \models I \to \iota\left(\mathrm{Subq}\big(Q, \sigma_{\mathrm{Consts}(\Sigma)} \cup \sigma_I, C\big)\right),$$

and as $\Phi_{Q,C}$ is the only formula containing $\mathrm{Subgoal}_{Q,C}$, we can remove it from the assumption so that

$$\begin{aligned} \Sigma \cup I &\models \iota\left(\mathrm{Subq}\big(Q, \sigma_{\mathrm{Consts}(\Sigma)} \cup \sigma_I, C\big)\right) \\ &= \mathrm{LRealz}_Q\big(\sigma_{\mathrm{Consts}(\Sigma)}, \sigma_I, \iota\big)\,. \end{aligned}$$

($\Longleftarrow$): We can perform all steps of ($\Longrightarrow$) in reverse order. $\square$

4.3.2 Glueing Subgoals

As a final step in the Datalog rewriting algorithm, we need to generate rules that combine subgoals to derive the final goal. To this end, we introduce the *goal atom* $\mathrm{Goal}_Q\left(\overrightarrow{\mathrm{FV}(Q)}\right)$ having $\mathrm{FV}(Q)$ as arguments.

Recall that a subgoal atom $\mathrm{Subgoal}_{Q,C}(\vec{t})$ indicates that the set C of bound variables are witnessed existentially with a partial substitution $\left\{\overrightarrow{\partial_Q C} \mapsto \vec{t}\right\}$ to the boundary of C. For a set $\mathrm{BVars} \supseteq \mathrm{FV}(Q)$ of variables, which we expect to be substituted by constants either in the base or in Σ, we define the following rule.

Definition 4.12. For a conjunctive query $Q = \exists \vec{z}.\, \bigwedge_{j \in J} A_j$ and a set $\mathrm{BVars} \supseteq \mathrm{FV}(Q)$, define the *subgoal glueing rule* $\mathrm{SglGlueingRule}_{\mathrm{BVars}}$ given by

$$\forall \overrightarrow{\mathrm{BVars}}.\left(\bigwedge_{C \in \mathcal{C}_{\mathrm{BVars}}} \mathrm{Subgoal}_{Q,C}\left(\overrightarrow{\partial_Q C}\right)\right) \wedge \mathrm{Atoms}_{\mathrm{BVars}} \to \mathrm{Goal}_Q\left(\overrightarrow{\mathrm{FV}(Q)}\right)$$

where $\mathrm{Atoms}_{\mathrm{BVars}}$ is the conjunction of atoms given by

$$\mathrm{Atoms}_{\mathrm{BVars}} = \bigwedge_{\substack{j \in J \\ \mathrm{Vars}(A_j) \subseteq \mathrm{BVars}}} A_j$$

and $\mathcal{C}_{\mathrm{BVars}}$ is the family of Q-connected components of $(\vec{z} \setminus \mathrm{BVars})$. $\square$

Example 4.13. Consider the query

$$\begin{aligned} Q = \exists z_1, z_4, z_5, z_6.\ & S(z_1, w_2) \wedge S(z_1, w_3) \wedge R(w_2, w_3) \\ & \wedge R(w_3, z_4) \wedge R(w_3, z_5) \wedge T(w_2, z_6, z_5). \end{aligned}$$

Q is similar to the query in Theorem 3.14, except except that we replaced z_2 and z_3 with free variables w_2 and w_3. If we let $\mathrm{BVars} = \{w_2, w_3, z_5\}$, then

$$\begin{aligned}\mathrm{SglGlueingRule}_{\mathrm{BVars}} = {} & \mathrm{Subgoal}_{Q,\{z_1\}}(w_2, w_3) \wedge \mathrm{Subgoal}_{Q,\{z_4\}}(w_3) \\ & \wedge \mathrm{Subgoal}_{Q,\{z_6\}}(w_2, w_3, z_5) \\ & \wedge R(w_2, w_3) \wedge R(w_3, z_5) \\ & \rightarrow \mathrm{Goal}_Q(w_2, w_3).\end{aligned}$$

Remark 4.14. Each $\mathrm{SglGlueingRule}_{\mathrm{BVars}}$ is "sound" (in a sense as in Theorem 4.11, by identifying subgoals with subquery fulfilments and the goal atom with query fulfilment), and also collectively complete (i.e. we can derive all answers to Q as goal facts if we can use all glueing rules) by Theorem 3.21.

4.3.3 Putting The Pieces Together

Finally, we combine components from Section 4.2, Section 4.3.1 and Section 4.3.2.

Algorithm 4 A rewriting procedure for GTGDs-CQ pairs

procedure REWRITE(Single-headed GTGDs Σ, Conjunctive query Q)
  result $\leftarrow \emptyset$
  result.addAll(AREW(Σ))
  for each $\mathcal{I} \in$ ENUMERATESUBQUERYENTAILMENTS(Σ, Q) **do**
    result.add(SubgoalRule($\mathcal{I}$))
  end for
  for each $\mathrm{FV}(Q) \subseteq \mathrm{BVars} \subseteq \mathrm{Vars}(Q)$ **do**
    result.add($\mathrm{SglGlueingRule}_{\mathrm{BVars}}$)
  end for
  return *result*
end procedure

Theorem 4.15. *Let Σ be a finite set of single-headed GTGDs and Q a conjunctive query. Then* REWRITE(Σ, Q) *in Algorithm 4 computes a Datalog rewriting of* (Σ, Q).

Proof. By correctness of Algorithm 3, Theorem 4.11 and Theorem 4.14. □

4.4 Optimising the Subquery Entailment Enumeration

When rewriting a GTGDs-CQ pair (Σ, Q) with Algorithm 4, the apparent bottleneck is the call ENUMERATESUBQUERYENTAILMENTS(Σ, Q) because we need to

- visit every (Σ, Q)-subquery entailment problem instance $\mathcal{I}$ and test if $\mathcal{I}$ satisfies Theorem 4.9, and
- descend a part of $\mathrm{SCTree}(I, \Sigma)$ (where I is the local instance in $\mathcal{I}$) and recursively search for a successful commit point.

Remark 4.16. As an illustrating example, consider the query

$$Q = \exists z_1, z_2, z_3.\ S(z_1, z_2) \wedge R(z_1, z_3).$$

We remark on the following inefficiencies with the naive implementation Algorithm 3 of EnumerateSubqueryEntailments:

1. *We are testing entailment relation for multiple isomorphic instances.* For example, consider the instance

   $$\mathcal{I}_1 = (C, I_1, \sigma_{\mathrm{Consts}(\Sigma)}, \sigma_I, \iota) = (\{z_1, z_2\}, \{S(1,2), T(1,1,2)\}, \emptyset, \{z_3 \mapsto 1\}, \emptyset).$$

   If we consider another instance

   $$\mathcal{I}_2 = (\{z_1, z_2\}, \{S(3,2), T(3,3,2)\}, \emptyset, \{z_3 \mapsto 3\}, \emptyset),$$

   it is clear that $\mathcal{I}_1$ is a subquery entailment if and only if $\mathcal{I}_2$ is because there is a renaming $\{1 \mapsto 3, 2 \mapsto 2\}$ of local names. If we have tested the entailment for $\mathcal{I}_1$, we need not repeat the process for $\mathcal{I}_2$.

2. Generalising (1), *we are testing entailment relation for subsumed instances.* For example, let $\mathcal{I}_1$ as in (1), and let

   $$\mathcal{I}_3 = (\{z_1, z_2\}, \{S(1,1), T(1,1,1), T(1,3,2), S(2,2)\}, \emptyset, \{z_3 \mapsto 1\}, \emptyset).$$

   If we already know that $\mathcal{I}_1$ is an entailment, then we may immediately conclude that $\mathcal{I}_3$ is also an entailment: The local instance I_3 of $\mathcal{I}_3$ is strictly *stronger than* I_1, by which we mean that there is a map $\sigma = \{1 \mapsto 1, 2 \mapsto 1\}$ from names in I_1 to names in I_3 such that for each $F \in I_1$, $\sigma(F) \in I_3$.

3. *We seem to be computing the same entailment problem repeatedly.* For example, consider the instance $\mathcal{I}_3$ from (2). The local realisation of $\mathcal{I}_3$ is $Q' = \exists z_1, z_2.\ S(z_1, z_2) \wedge R(z_1, 1)$ since z_3 is mapped to 1. The call

   $$\textsc{EntailsConnectedBCQ}(I_3, \Sigma, Q'),$$

   may first try IsSuccessfulCommitPoint with the root instance I_3. Within IsSuccessfulCommitPoint, we might guess a commit map $\{z_2 \mapsto 3\}$ and check if this is a good guess by calling

   $$\textsc{BooleanQSatisfiedWith}(I_3, \Sigma, Q', \{z_2 \mapsto 3\}).$$

   As a result, we will make a recursive call

   $$\textsc{EntailsConnectedBCQ}(I_3, \Sigma, \exists z_1.S(z_1, 3) \wedge R(z_1, 1)).$$

But this is exactly the first call to EntailsConnectedBCQ when we decide if

$$(\{z_1\}, I_3, \emptyset, \{z_2 \mapsto 3, z_3 \mapsto 1\}, \emptyset)$$

is an entailment.

□

In the remainder of this chapter, we describe three optimisation techniques: two that address the issue (3) in Section 4.4.1 and in Section 4.4.2, and the other that partially resolves (1) in Section 4.4.3. However, we leave (2) as a problem for future work.

4.4.1 Dynamic Programming

As observed in Theorem 4.16, the issue is that

- even though we are working with locally realised subqueries, we are essentially deciding split subquery entailment instances, yet
- we are not making use of the result of past invocations.

The obvious solution is to work directly with subquery entailment instances instead, memoise all past results in a hash map (which we call the *DP table*), fill in the hash map from smaller (i.e. instances with smaller C) instances and finally output all results at the end of EnumerateSubqueryEntailments. We call this optimisation the *dynamic programming optimisation*.

To proceed, we need to define the process of "splitting" a subquery entailment problem instance, a process which is unsurprisingly similar to Theorem 3.18.

Definition 4.17. Let Q be a conjunctive query, $\mathcal{I} = (C, I, \sigma_{\mathrm{Consts}(\Sigma)}, \sigma_I, \iota)$ a (Σ, Q)-subquery entailment problem instance and $\sigma_{\mathrm{commit}} : C \rightharpoonup \mathrm{LNames}(I)$ a partial map.

The *committed part* $\mathrm{Commq}(\mathcal{I}, \sigma_{\mathrm{commit}})$ *of* $\mathcal{I}$ *according to* σ_{commit} is the variable-free query $\mathrm{Commq}\big(\mathrm{LRealz}_Q\big(\sigma_{\mathrm{Consts}(\Sigma)}, \sigma_I, \iota\big), \sigma_{\mathrm{commit}}\big)$.

For each Q-connected component C' of $(C \setminus \mathrm{dom}(\sigma_{\mathrm{commit}}))$, the *subinstance* $\mathrm{Subi}(\mathcal{I}, \sigma_{\mathrm{commit}}, C')$ *of* $\mathcal{I}$ *induced by* σ_{commit} *and* C' is the (Σ, Q)-subquery entailment problem instance defined by

$$\mathrm{Subi}(\mathcal{I}, \sigma_{\mathrm{commit}}, C') = (C', I, \sigma'_{\mathrm{Consts}(\Sigma)}, \sigma'_I, \iota'),$$

where

$$\begin{aligned} \sigma'_{\mathrm{Consts}(\Sigma)} &= \sigma_{\mathrm{Consts}(\Sigma)} \upharpoonright \partial_Q C', \\ \sigma'_I &= (\sigma_I \cup \sigma_{\mathrm{commit}}) \upharpoonright \partial_Q C', \\ \iota' &= \iota \upharpoonright S_{Q,C'}, \end{aligned}$$

and $S_{Q,C'}$ defined as in Theorem 4.5.

□

Algorithm 5 Recursive Subquery Entailment Enumeration

// decides if $\mathcal{I}$ is a (Σ, Q)-subquery entailment
// we can optionally cache the answer of this function using a hash map
procedure IsSubqueryEntailment($\mathcal{I} = (C, I, \sigma_{\mathrm{Consts}(\Sigma)}, \sigma_I, \iota)$)
  for each local instance I' in $\mathrm{SCTree}(I, \Sigma)$ that can be reached without dropping local names $\mathrm{range}(\sigma_I) \cup \mathrm{range}(\iota)$ **do**
    if Splits($(C, I', \sigma_{\mathrm{Consts}(\Sigma)}, \sigma_I, \iota)$) **then**
      return true
    end if
  end for
  return false
end procedure

// decides if $\mathcal{I}$ splits into subquery entailments
procedure Splits($\mathcal{I} = (C, I, \sigma_{\mathrm{Consts}(\Sigma)}, \sigma_I, \iota)$)
  for each nonempty $\sigma_{\mathrm{commit}} : C \rightharpoonup \mathrm{LNames}(I)$ **do**
    if SplitsWith($\mathcal{I}$, σ_{commit}) **then**
      return true
    end if
  end for
  return false
end procedure

// decides if $\mathcal{I}$ splits into subquery entailments
// with commit map $\sigma_{\mathrm{commit}} : \mathrm{Vars}(Q) \rightharpoonup \mathrm{LNames}(I)$
procedure SplitsWith($\mathcal{I} = (C, I, \sigma_{\mathrm{Consts}(\Sigma)}, \sigma_I, \iota)$, σ_{commit})
  $I_{\mathrm{Dsat}} \leftarrow$ DatalogSaturate($\mathrm{ARew}(\Sigma)$, I)
  baseSatisfied $\leftarrow I_{\mathrm{Dsat}} \models \mathrm{Commq}(\mathcal{I}, \sigma_{\mathrm{commit}})$
  $C_1, \ldots, C_n \leftarrow Q$-connected components of $C \setminus \mathrm{dom}(\sigma_{\mathrm{commit}})$
  allComponentsSatisfied $\leftarrow \bigwedge_{i=1}^{n}$
    IsSubqueryEntailment($\mathrm{Subi}(\mathcal{I}, \sigma_{\mathrm{commit}}, C_i)$)
  return baseSatisfied **and** allComponentsSatisfied
end procedure

procedure EnumerateSubqueryEntailments(
  finite set Σ of single-headed GTGDs, conjunctive query Q)
  for each (Σ, Q)-subquery entailment problem instance $\mathcal{I}$ **do**
    if IsSubqueryEntailment($\mathcal{I}$) **then**
      output $\mathcal{I}$
    end if
  end for
end procedure

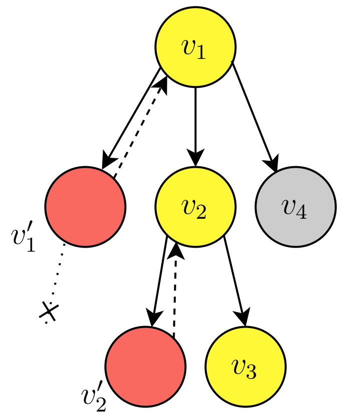

Figure 4.4: A state during a DFS of the chase tree. Red nodes ($\{v_1', v_2'\}$) indicate unsuccessful subquery entailment instances. We had already seen v_1' before this search, so we terminated the fruitless search below v_1'. Yellow nodes are the instances whose entailment we are not yet certain. We explore the grey node v_4 only if we give up the search below v_2.

Algorithm 5 is another implementation of ENUMERATESUBQUERYENTAILMENTS based on Theorem 4.17. The algorithm is equivalent to Algorithm 3, but since IS-SUBQUERYENTAILMENT is a Boolean function over (Σ, Q)-subquery entailment problem instances, we can memoise answers of ISSUBQUERYENTAILMENT and avoid recomputations.

4.4.2 DFS Optimisation

As discussed briefly at the end of Chapter 3, instead of computing the set of all local instances I' satisfying the condition at line 4 of Algorithm 5, we can use a depth-first search (DFS) to explore the space of all such local instances.

During this process of deciding whether $\mathcal{I} = (C, I, \sigma_{\mathrm{Consts}(\Sigma)}, \sigma_I, \iota)$ is a subquery entailment, each instance I' we encounter in the DFS corresponds to a problem instance $\mathcal{I}' = (C, I', \sigma_{\mathrm{Consts}(\Sigma)}, \sigma_I, \iota)$. If we use the DP optimisation, we can check if we have already seen $\mathcal{I}'$, and if so, immediately stop exploring the chase tree. We illustrate this process in Figure 4.4.

If we find a node v that admits a splitting into subquery entailments, we mark all ancestors of v as **true**, and if the exploration below a node v is unsuccessful, we mark v **false**, move up and try siblings of v. We show this marking process in Figure 4.5.

We call this efficient traversal of the chase tree the *DFS optimisation*. Combined with the DP optimisation, we never need to compute SPLITS twice for the same instance.

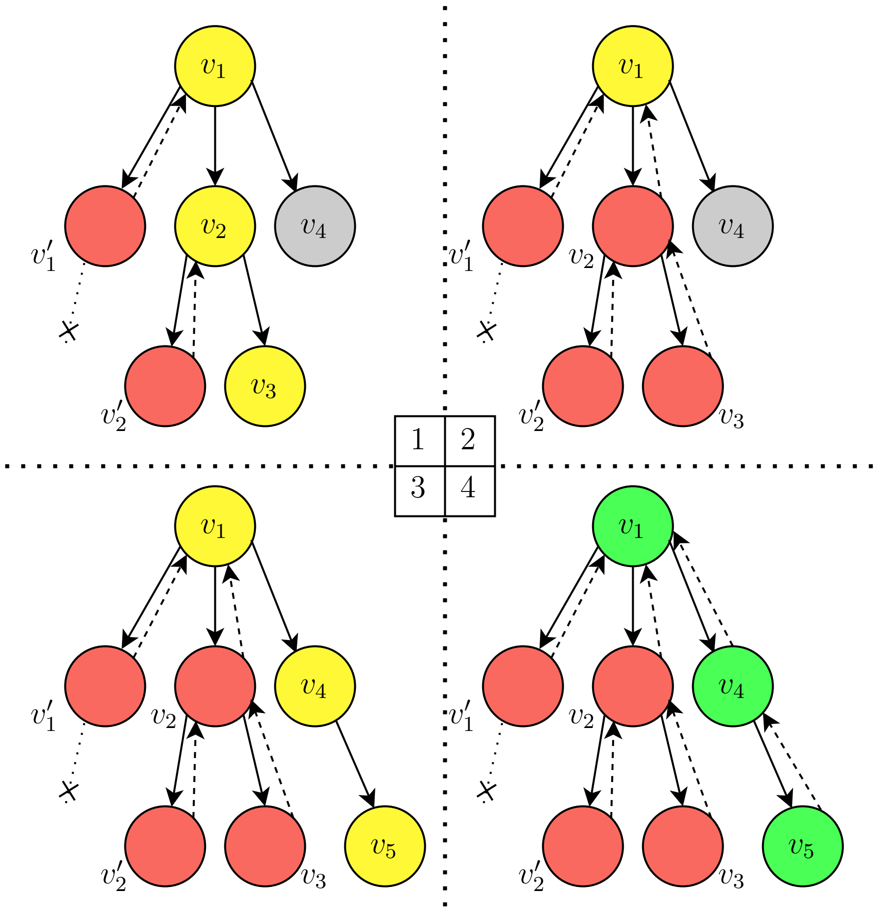

Figure 4.5: The marking process in chase tree DFS. If we start from the state on the top-left and find out that v_3 does not admit a splitting, we mark v_3 false since v_3 has no children. We move up to v_2, but as v_2 has no unexplored children, we mark v_2 false too (top-right). We try the sibling node v_4 of v_2 and descend further to v_5 (bottom-left). Finally, v_5 admits a splitting, so we mark all ancestors of v_5 true (bottom-right).

4.4.3 Instance Normalisation

Finally, we address the first point of Theorem 4.16 concerning isomorphic instances.

Recall that we use $2K$ local names as the set of K names is insufficient to maintain a tree-like structure of local instances (Figure 4.1).

However, in Algorithm 5, even with DP and DFS optimisations in place, we only partially use the structure between local instances when we descend the chase tree. All we require is that the local names in $\mathrm{range}(\sigma_I) \cup \mathrm{range}(\iota)$ be preserved. In particular, we do not care if a local name (not in $\mathrm{range}(\sigma_I) \cup \mathrm{range}(\iota)$) at a node v is introduced at v or is inherited from the parent of v.

This observation implies that, during the DFS, we are free to "normalise" local instances as long as we fix the names in $\mathrm{range}(\sigma_I) \cup \mathrm{range}(\iota)$. Since at most K local names are active in a local instance, we can always restrict ourselves to local names from $\{1, \ldots, K\}$. We call the following strategy the *instance normalisation*:

- In the top-level function EnumerateSubqueryEntailments, we only enumerate local instances over local names from $\{1, \ldots, K\}$.
- Within IsSubqueryEntailment, we always re-map local names in chased instances to the range $\{1, \ldots, K\}$, provided that we fix all names in $\mathrm{range}(\sigma_I) \cup \mathrm{range}(\iota)$.
- Consequently, the DP table only remembers entailment relations for instances over $\{1, \ldots, K\}$.

Remark 4.18. Combining all optimisations so far, we obtain the following performance characteristic.

Suppose for simplicity that Q only contains constants from Σ. Let $\mathrm{sig} = \mathrm{Predicates}(\Sigma \cup \{Q\})$, $K = \max_{P \in \mathrm{sig}} \mathrm{Arity}(P)$. Then there are $K + |\mathrm{Consts}(\Sigma)|$ terms we use in a local instance. Suppose further that all $P \in \mathrm{sig}$ have arity K. Then there are $|\mathrm{sig}|(K + |\mathrm{Consts}(\Sigma)|)^K$ facts that we can form. Therefore there are $2^{\left(|\mathrm{sig}|(K+|\mathrm{Consts}(\Sigma)|)^K\right)}$ local instances for which we need to decide subquery entailments. Since there are at most $2^{|\mathrm{Vars}(Q)|}$ connected variable sets, each of which has at most $(K + |\mathrm{Consts}(\Sigma)|)^{|\mathrm{Vars}(Q)|}$ different realisations, we estimate to test at most

$$
\begin{aligned}
&2^{\left(|\mathrm{sig}|(K+|\mathrm{Consts}(\Sigma)|)^K\right)} \times 2^{|\mathrm{Vars}(Q)|} \times (K + |\mathrm{Consts}(\Sigma)|)^{|\mathrm{Vars}(Q)|} \\
&\qquad = 2^{\left(|\mathrm{sig}|(K+|\mathrm{Consts}(\Sigma)|)^K + |\mathrm{Vars}(Q)|\right)} \times (K + |\mathrm{Consts}(\Sigma)|)^{|\mathrm{Vars}(Q)|} \qquad (4.1)
\end{aligned}
$$

instances for subquery entailment. Each instance is tested only once (by DP and DFS optimisations), and each test takes time exponential to $|\mathrm{ARew}(\Sigma)|$ due to Datalog-saturating local instances.

Overall, the algorithm produces a rewriting in time *always* doubly exponential to the maximum arity. □

Chapter 5

Implementation and Testing

We built an open-sourced prototypical rewriting library in Java, combining all ideas from Chapter 3 and Chapter 4. On top of the core library, the system has a command-line interface where an end-user can interact with the rewriter.

5.1 Architecture

Our implementation sits on top of two libraries: *Guarded-saturation* [5], which provides implementations for computing atomic rewritings, and its dependency `pdq-common-2.0.0`, which provides a foundation for expressing first-order formulae.

The implementation of the rewriter closely follows Algorithm 4. For EnumerateSubqueryEntailments, we provide three implementations based on Algorithm 5, which respectively have (*DP*), (*DP + normalisation*) and (*DP + normalisation + DFS*) optimisations applied.

Further, to minimise the evaluation time of the output Datalog program, we apply *rule subsumptions* at the end: If we obtained two rules $\tau_1 = \beta_1 \to \eta_1$ and $\tau_2 = \beta_2 \to \eta_2$, and there exists a homomorphism $\sigma : \beta_1 \to \beta_2$ such that $\sigma(\eta_1) \supseteq \sigma(\eta_2)$, then we discard τ_2 in favour of τ_1.

We illustrate the overall architecture in Figure 5.1.

5.2 Correctness Tests

The codebase contains a naive Datalog engine, a (nested-loop) join algorithm, output rule subsumption and many other utility components, all of which are extensively property-based-tested with ScalaCheck [10]. In addition, the overall rewriting system is integration-tested with JUnit 5 [11].

Since there are no other implementations for GTGDs-CQ rewriting, we could not compare our algorithm to existing implementations on general queries. However, we can translate an “acyclic” existential query into an atomic query by adding a few guarded rules. For instance, the query $Q = \exists x, y. R(w, x) \wedge S(y, x)$

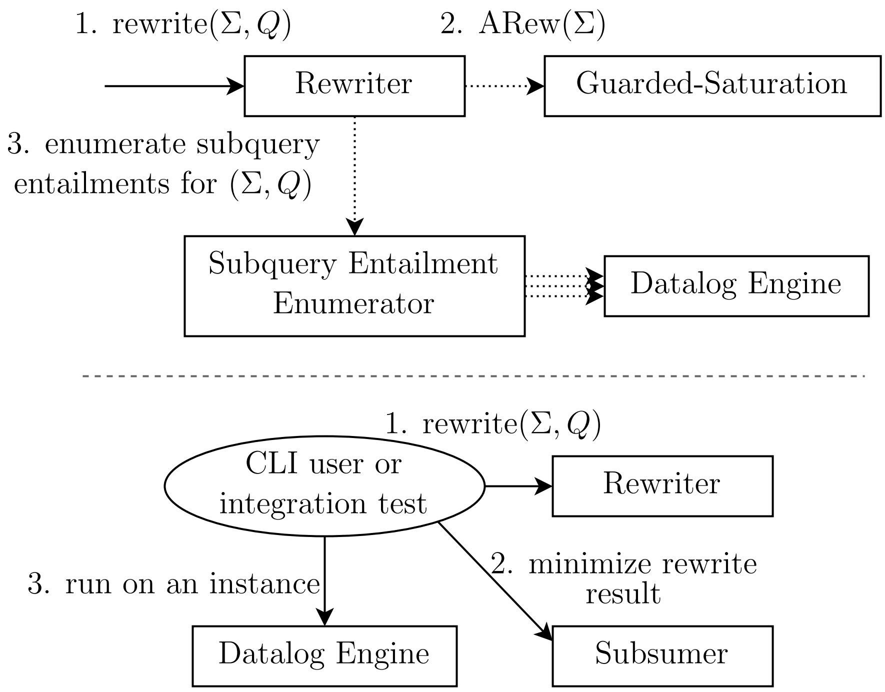

Figure 5.1: The overall architecture of the rewriting system. (Top) The rewriter is a composition of an atomic rewriter with an implementation of ENUMERATE-SUBQUERYENTAILMENTS. Every dotted line indicates a call through an interface. The architecture is flexible in that one can independently apply optimisations to one or more components. (Bottom) When a CLI user or an integration test performs a rewriting, we minimise the result using subsumption.

is existential, but if we add rules

$$\forall x, y.\ S(y, x) \to \mathrm{Goal}_1(x)$$
$$\forall w, x.\ R(w, x) \wedge \mathrm{Goal}_1(x) \to \mathrm{Goal}(w)$$

where Goal_1 and Goal are fresh, then Q and $\mathrm{Goal}(w)$ are equivalent queries.

With this observation, we manually translated a few acyclic queries, rewritten both the translated and the original rule sets using *Guarded-saturation* and our rewriting system, and finally tested if two rewritings agreed when run on randomly generated instances.

5.3 Example Runs

Figure 5.2 is an example of an interaction between a user and the application where the user attempts to rewrite

$$Q = \exists y.R(c_1, y) \wedge R(y, w) \wedge R(w, c_3)$$

```
> add-rule R(x_1, c_1), P(x_1) -> EE y_1. R(c_1, y_1), R(y_1, x_1), P(y_1)
Registered rule: R(x_1,c_1) & P(x_1) -> EE y_1. R(y_1,x_1) & P(y_1) & R(c_1,y_1)
> add-rule R(c_1, x_1) -> R(x_1, c_1), P(x_1)
Registered rule: R(c_1,x_1) -> R(x_1,c_1) & P(x_1)
> rewrite dfs-normalizing EE y. R(c_1, y), R(y, w), R(w, c_3)
Rewriting query:(exists[y]R(c_1,y) & R(y,w) & R(w,c_3))
  using GuardedRuleAndQueryRewriter{...}, with registered rules:
  R(x_1,c_1) & P(x_1) -> EE y_1. R(y_1,x_1) & P(y_1) & R(c_1,y_1)
  R(c_1,x_1) -> R(x_1,c_1) & P(x_1)

(GSat logs)

Done rewriting query in 3008940600 nanoseconds.
Minimizing the result...
# of subgoal derivation rules in original output: 1895
# of subgoal derivation rules in minimalExactBodyMinimizedRewriting: 22
# of subgoal derivation rules in minimizedRewriting: 7
Done minimizing the result in 94796800 nanoseconds.
Rewritten query:
  Goal atom: IP0_GOAL(w)
  Atomic rewriting part:
    R(GSat_u1,GSat_u2), R(c_1,GSat_u1), P(GSat_u1) :- IP0_NI_0(GSat_u2,GSat_u1)
    R(GSat_u1,c_1), P(GSat_u1) :- R(c_1,GSat_u1)
  Subgoal derivation part:
    IP0_SQ0_SGL_0(x_1) :- R(x_2,x_1), R(c_1,x_2), R(x_2,c_1), P(x_2)
    IP0_SQ0_SGL_0(x_0) :- R(c_1,x_0)
    IP0_SQ0_GOAL(w) :- IP0_SQ0_SGL_0(w)
    IP0_SQ0_SGL_0(c_1) :- R(x_1,c_1), P(x_1)
    IP0_SQ0_GOAL(w) :- R(c_1,y), R(y,w)
    IP0_SQ0_SGL_0(x_0) :- R(x_0,c_1), P(x_0)
    IP0_GOAL(w) :- R(w,c_3), IP0_SQ0_GOAL(w)
```

Figure 5.2: An interaction with the command-line interface. The user uses the algorithm with all three optimisations applied.

under rules

$$\forall x_1.\ R(x_1, c_1) \land P(x_1) \to \exists y_1.\ R(c_1, y_1) \land R(y_1, x_1) \land P(y_1),$$
$$\forall x_1.\ R(c_1, x_1) \to R(x_1, c_1) \land P(x_1).$$

The experiment is performed on a Windows 11 computer with an Intel Core i9-9900 CPU @ 3.10 GHz and 64 GB of RAM.

For comparison, with only (DP) and (DP + normalisation) optimisations applied, rewriting the same input as in Figure 5.2 takes about 7.3 seconds and 160 seconds respectively, demonstrating the effectiveness of optimisations in Section 4.4.

Chapter 6

Conclusions and Further Discussion

With the help of atomic rewritings, we revised the theory of GTGD chases by introducing the notion of *shortcutting chase trees*. Furthermore, we observed how a query homomorphism is decomposed in a shortcutting chase tree and devised a recursive decision procedure based on the observation. By further analysing the structure of the chase tree, we showed that *local instances* can convey how queries are satisfied. Based on this analysis, we developed a rewriting algorithm employing the notion of *subquery entailments*.

We discussed issues encountered by a naive algorithm for enumerating subquery entailments and provided a few optimisation techniques. Finally, we implemented the optimised version of the algorithm, which can handle arity-2 rules and a few constants in the rule and the query.

6.1 Limitations and Future Work

The current implementation lacks optimisations for trimming down the space of subquery entailment problem instances. Instead, it always explores the whole space, whose size is doubly exponential in the maximum arity of the input signature and exponential in the number of constants and predicates (Theorem 4.18), making it impractical to rewrite large inputs such as real-world ontologies. Even though this matches with the theoretical lower bound of query answering procedure (which is 2EXPTIME for arbitrary arity and EXPTIME for bounded arity [6]), we may be able to overcome this issue in some cases by analysing the structure of input rules. For instance, if a rule constant c only appears in heads and not in the query, it is redundant to consider local instances containing facts with c since no rule requires such facts.

Arguably, the most crucial optimisation is handling instance subsumption, as discussed in Theorem 4.16: If a single atom $R(1, 2)$ suffices to entail a subquery, *all* local instances containing a fact with R no longer need to be tested for entailment, reducing the search space by a factor of $16 = 2^4$. We leave for future work the method for efficiently controlling the search space.

Another performance consideration is, as remarked in Section 5.2, that we can rewrite some queries into atomic queries by adding a few guarded rules. Our system does not perform such preprocessing, nor does it reduce subquery entailment problems to atomic queries, even when induced subqueries are acyclic. Investigating the effectiveness of such input transformation is left for future work.

Moreover, our prototypical system spends most of its CPU time in Datalog-saturating and chasing the local instances. We use an inefficient join algorithm without indexes and the Naive Evaluation to saturate instances for simplicity. One might want to incorporate more sophisticated join and saturation algorithms and compare their performances.

Finally, as mentioned in the cited paper, the result in [2] concerning rewritability extends to a slightly wider class of TGDs known as frontier-guarded TGDs, where only frontier-variables have to be guarded in the body. Therefore, it is of theoretical interest if we could extend our approach to this class of TGDs.

Bibliography

[1] Francois Bancilhon and Raghu Ramakrishnan. “An Amateur’s Introduction to Recursive Query Processing Strategies”. In: *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’86. Washington, D.C., USA: Association for Computing Machinery, 1986, pp. 16–52. ISBN: 0897911911. DOI: 10.1145/16894.16859. URL: https://doi.org/10.1145/16894.16859.

[2] Vince Bárány, Michael Benedikt, and Balder ten Cate. “Rewriting Guarded Negation Queries”. In: *Mathematical Foundations of Computer Science 2013*. Ed. by Krishnendu Chatterjee and Jirí Sgall. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 98–110. ISBN: 978-3-642-40313-2.

[3] C. Beeri and M. Y. Vardi. “The implication problem for data dependencies”. In: *Automata, Languages and Programming*. Ed. by Shimon Even and Oded Kariv. Berlin, Heidelberg: Springer Berlin Heidelberg, 1981, pp. 73–85. ISBN: 978-3-540-38745-9.

[4] M. Benedikt, M. Buron, S. Germano, K. Kappelmann, and B. Motik. “Rewriting the infinite chase”. In: *Proceedings of the VLDB Endowment* 15.11 (July 2022), pp. 3045–3057. DOI: 10.14778/3551793.3551851.

[5] Michael Benedikt, Maxime Buron, Stefano Germano, Kevin Kappelmann, and Boris Motik. *Rewriting the Infinite Chase*. Version 1.0.0. URL: https://github.com/KRR-Oxford/Guarded-saturation.

[6] A. Calì, G. Gottlob, and M. Kifer. “Taming the infinite chase: Query answering under expressive relational constraints”. In: *Journal of Artificial Intelligence Research* 48 (2013), pp. 115–174. DOI: 10.1613/jair.3873.

[7] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. “Complexity and Expressive Power of Logic Programming”. In: *ACM Comput. Surv.* 33.3 (Sept. 2001), pp. 374–425. ISSN: 0360-0300. DOI: 10.1145/502807.502810. URL: https://doi.org/10.1145/502807.502810.

[8] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. “Data exchange: semantics and query answering”. In: *Theoretical Computer Science* 336.1 (2005). Database Theory, pp. 89–124. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2004.10.033. URL: https://www.sciencedirect.com/science/article/pii/S030439750400725X.

[9] K. Kappelmann. *Decision Procedures for Guarded Logics*. Version 2. Nov. 9, 2019. DOI: https://doi.org/10.48550/arXiv.1911.03679. arXiv: 1911.03679v2.

[10] Rickard Nilsson. *ScalaCheck: Property-based testing for scala*. https://scalacheck.org/. Accessed: 2023-05-14. 2021.

[11] The JUnit Team. *JUnit 5*. https://junit.org/junit5/. Accessed: 2023-05-14. 2023.