The Space-Efficient Core of Vadalog

GERALD BERGER, TU Wien, Austria
GEORG GOTTLOB, University of Oxford, UK and TU Wien, Austria
ANDREAS PIERIS, University of Edinburgh, UK and University of Cyprus, Cyprus
EMANUEL SALLINGER, University of Oxford, UK and TU Wien, Austria

Vadalog is a system for performing complex reasoning tasks such as those required in advanced knowledge graphs. The logical core of the underlying Vadalog language is the warded fragment of tuple-generating dependencies (TGDs). This formalism ensures tractable reasoning in data complexity, while a recent analysis focusing on a practical implementation led to the reasoning algorithm around which the Vadalog system is built. A fundamental question that has emerged in the context of Vadalog is whether we can limit the recursion allowed by wardedness in order to obtain a formalism that provides a convenient syntax for expressing useful recursive statements, and at the same time achieves space-efficiency. After analyzing several real-life examples of warded sets of TGDs provided by our industrial partners, as well as recent benchmarks, we observed that recursion is often used in a restricted way: the body of a TGD contains at most one atom whose predicate is mutually recursive with a predicate in the head. We show that this type of recursion, known as piece-wise linear in the Datalog literature, is the answer to our main question. We further show that piece-wise linear recursion alone, without the wardedness condition, is not enough as it leads to undecidability. We also study the relative expressiveness of the query languages based on (piece-wise linear) warded sets of TGDs. Finally, we give preliminary experimental evidence for the practical effect of piece-wise linearity on Vadalog.

ACM Reference Format:
Gerald Berger, Georg Gottlob, Andreas Pieris, and Emanuel Sallinger. 2022. The Space-Efficient Core of Vadalog. *ACM Trans. Datab. Syst.* 1, 1 (March 2022), 46 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

In recent times, thousands of companies world-wide wish to manage their own knowledge graphs (KGs), and are looking for adequate knowledge graph management systems (KGMS). The term knowledge graph originally only referred to Google's Knowledge Graph, i.e., "a knowledge base used by Google and its services to enhance its search engine's results with information gathered from a variety of sources.[1]" In the meantime, several other large companies have constructed their own knowledge graphs, and many more companies would like to maintain a private corporate knowledge graph incorporating large amounts of data in form of database facts, both from corporate and public sources, as well as rule-based knowledge. Such a corporate knowledge graph is expected to contain relevant business knowledge, for example, about customers, products, prices, and competitors, rather than general knowledge from Wikipedia and similar sources. It should be managed by a KGMS, that is, a knowledge base management system that performs complex rule-based reasoning

[1]https://en.wikipedia.org/wiki/Knowledge_Graph

Authors' addresses: Gerald Berger, TU Wien, Austria, georg.gottlob@cs.ox.ac.uk; Georg Gottlob, University of Oxford, UK, TU Wien, Austria, georg.gottlob@cs.ox.ac.uk; Andreas Pieris, University of Edinburgh, UK, University of Cyprus, Cyprus, apieris@inf.ed.ac.uk; Emanuel Sallinger, University of Oxford, UK, TU Wien, Austria, emanuel.sallinger@cs.ox.ac.uk.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
0362-5915/2022/3-ART $15.00
https://doi.org/0000001.0000001

tasks over very large amounts of data and, in addition, provides methods and tools for data analytics and machine learning [8].

1.1 The Vadalog System

Vadalog is a system for performing complex reasoning tasks such as those required in advanced knowledge graphs [9]. It is Oxford's contribution to the VADA research project,[2] a joint effort of the universities of Oxford, Manchester, and Edinburgh, as well as around 20 industrial partners such as Facebook, BP, and the NHS (UK national health system). One of the most fundamental reasoning tasks performed by Vadalog is *ontological query answering*: given a database D, an ontology Σ (which is essentially a set of logical assertions that allow us to derive new intensional knowledge from D), and a query $q(\bar{x})$ (typically a conjunctive query), the goal is to compute the certain answers to q w.r.t. the knowledge base consisting of D and Σ, i.e., the tuples of constants $\bar{c}$ such that, for every relational instance $I \supseteq D$ that satisfies Σ, I satisfies the Boolean query $q(\bar{c})$ obtained after instantiating $\bar{x}$ with $\bar{c}$. Due to Vadalog's ability to perform ontological query answering, it is currently used as the core deductive database component of the overall Vadalog KGMS, as well as at various industrial partners including the finance, security, and media intelligence industries.

The logical core of the underlying Vadalog language is a rule-based formalism known as *warded Datalog*$^{\exists}$ [18], which is a member of the Datalog$^{\pm}$ family of knowledge representation languages [15]. Warded Datalog$^{\exists}$ generalizes Datalog with existential quantification in rule heads, and at the same time applies a restriction on how certain "dangerous" variables can be used; details are given in Section 3. Such a restriction is needed as basic reasoning tasks, e.g., ontological query answering, under arbitrary Datalog$^{\exists}$ rules become undecidable; see, e.g., [7, 14]. Let us clarify that Datalog$^{\exists}$ rules are essentially *tuple-generating dependencies* (TGDs) of the form $\forall\bar{x}\forall\bar{y}(\varphi(\bar{x},\bar{y}) \rightarrow \exists\bar{z}\,\psi(\bar{x},\bar{z}))$, where φ (the *body*) and ψ (the *head*) are conjunctions of atoms. Therefore, knowledge representation and reasoning should be seen as a modern application of TGDs, which have been introduced decades ago as a unifying framework for database integrity constraints.

The key properties of warded Datalog$^{\exists}$, which led to its adoption as the logical core on top of which the Vadalog language is built, can be summarized as follows:

(1) *Recursion over KGs.* It is able to express full recursion and joins, needed to express complex reasoning tasks over KGs. Moreover, navigational capabilities, empowered by recursion, are vital for graph-based structures.
(2) *Ontological Reasoning over KGs.* After adding a very mild and easy to handle negation, the language is able to express SPARQL reasoning under the OWL 2 QL entailment regime. Recall that SPARQL is the standard language for querying the Semantic Web,[3] while OWL 2 QL is a prominent profile of the OWL 2 Web Ontology Language, the standard formalism for modeling Semantic Web ontologies.[4]
(3) *Low Complexity.* Reasoning, in particular, ontological query answering, is tractable (in fact, polynomial time) in data complexity, which is a minimal requirement for allowing scalability over large volumes of data.

Warded Datalog$^{\exists}$ turned out to be powerful enough for expressing all the tasks given by our industrial partners, while a recent analysis of it focusing on a practical implementation led to the reasoning algorithm around which the Vadalog system is built [9].

[2] http://vada.org.uk/
[3] http://www.w3.org/TR/rdf-sparql-query
[4] https://www.w3.org/TR/owl2-overview/

1.2 Research Challenges

With the aim of isolating more refined formalisms, which will lead to yet more efficient reasoning algorithms, the following fundamental question has emerged in the context of Vadalog:

Can we limit the recursion allowed by wardedness in order to obtain a formalism that provides a convenient syntax for expressing useful statements, importantly, most of the scenarios provided by our partners, and at the same time achieves space-efficiency, in particular, NLogSpace *data complexity?*

Let us stress that NLogSpace data complexity is the best that we can hope for since navigational capabilities are vital for graph-based structures, and already graph reachability is NLogSpace-hard. It is known that NLogSpace is contained in the class NC_2 of highly parallelizable problems. This means that reasoning in the more refined formalism for which we are aiming is principally parallelizable, unlike warded Datalog$^\exists$, which is PTime-complete and intrinsically sequential. Our ultimate goal is to exploit this in the future for the parallel execution of reasoning tasks in both multi-core settings and in the map-reduce model. In fact, we are currently in the process of implementing a multi-core implementation for the refined formalism proposed by the present work.

Extensive benchmark results are available for the Vadalog system, based on a variety of synthetic and industrial scenarios, including the following:

- ChaseBench [10], a benchmark that targets data exchange and query answering problems.
- iBench, a data exchange benchmark developed at the University of Toronto [4].
- iWarded, a benchmark specifically targeted at warded sets of TGDs.
- A benchmark based on DBpedia, and a number of other synthetic and industrial scenarios [9].

Note that all the above benchmarks consist of warded sets of TGDs. In fact, a good part of them are not *warded by chance*, i.e., they contain joins among "harmful" variables, which is one of the distinctive features of wardedness [9]. In the conference paper [11], on which the present work is based, after analyzing the above benchmarks, it has been observed that recursion is often used in a restricted way. Approximately 70% of the TGD-sets use recursion as follows: the body of a TGD contains at most one atom whose predicate is mutually recursive with a predicate in the head. Actually, approximately 55% of the TGD-sets directly use the above type of recursion, while 15% can be transformed into warded sets of TGDs that use recursion as above. This relies on a standard elimination procedure of unnecessary non-linear recursion. For example,

$$\forall x \forall y (E(x,y) \rightarrow T(x,y)) \qquad \forall x \forall y \forall z (T(x,y) \wedge T(y,z) \rightarrow T(x,z)),$$

which compute the transitive closure of E using non-linear recursion, can be rewritten as

$$\forall x \forall y (E(x,y) \rightarrow T(x,y)) \qquad \forall x \forall y \forall z (E(x,y) \wedge T(y,z) \rightarrow T(x,z)),$$

that uses linear recursion. Interestingly, the type of recursion discussed above has been already studied in the context of Datalog, and is known as *piece-wise linear*; see, e.g., [1]. It is a refinement of the well-known *linear* recursion [23, 24], already mentioned in the above example, which allows only one intensional predicate (i.e., a predicate that occurs in the head of at least one rule) to appear in the body, while all the other predicates are extensional (i.e., predicates that are not intensional).

Based on this key observation, the following research questions have immediately emerged:

(1) Does warded Datalog$^\exists$ with piece-wise linear recursion achieve space-efficiency for query answering?[5]
(2) Is the combination of wardedness and piece-wise linearity justified? That is, can we achieve the same space efficiency with piece-wise linear Datalog$^\exists$ without the wardedness condition?

[5]The idea of combining wardedness with piece-wise linearity has been already mentioned in the invited paper [8], while the obtained formalism is called strongly warded.

(3) What is the expressiveness of the query language based on warded Datalog$^{\exists}$ with piece-wise linear recursion relative to prominent languages such as Datalog?

The above questions have been already posed and studied in the conference paper [11] with the aim of obtaining new insights towards more efficient reasoning algorithms, in particular, towards parallel execution of reasoning tasks. In fact, the goal of [11] was to analyze piece-wise linearity, and provide definitive answers to the above questions. The present work extends and improves [11] in several ways; more details are given below.

1.3 Summary of Contributions

Our main results can be summarized as follows:

(1) In Section 4 we show that ontological query answering under warded Datalog$^{\exists}$ with piece-wise linear recursion is NLogSpace-complete in data complexity, and PSpace-complete in combined complexity, which provides a definitive answer to our first question. This is a rather involved result that relies on a novel notion of resolution-based proof tree, which is of independent interest. In particular, we show that ontological query answering under warded Datalog$^{\exists}$ with piece-wise linear recursion boils down to the problem of checking whether a proof tree that enjoys certain properties exists, which in turn can be done via a space-bounded non-deterministic algorithm. Interestingly, our machinery allows us to re-establish the complexity of ontological query answering under warded Datalog$^{\exists}$ via an algorithm that is significantly simpler than the one employed in [18]. Our algorithm is essentially the non-determinisitc algorithm for piece-wise linear warded Datalog$^{\exists}$ with the crucial difference that it employs alternation.
(2) To our surprise, ontological query answering under piece-wise linear Datalog$^{\exists}$, without the wardedness condition, is undecidable. This result, which is shown in Section 5 via a reduction from the unbounded tiling problem, provides a definitive answer to our second question: the combination of wardedness and piece-wise linearity is indeed justified.
(3) In Section 6 we investigate the relative expressive power of the query language based on warded Datalog$^{\exists}$ with piece-wise linear recursion, which consists of all the queries of the form $Q = (\Sigma, q)$, where Σ is a warded set of TGDs with piece-wise linear recursion, and q is a conjunctive query, while the evaluation of Q over a database D is precisely the certain answers to q w.r.t. D and Σ. By exploiting our novel notion of proof tree, we show that it is equally expressive to piece-wise linear Datalog. The same approach allows us to elucidate the relative expressiveness of the query language based on warded Datalog$^{\exists}$ (with arbitrary recursion), showing that it is equally expressive to Datalog. We also adopt the more refined notion of program expressive power, introduced in [2], which aims at the decoupling of the set of TGDs and the actual conjunctive query, and show that the query language based on warded Datalog$^{\exists}$ (with piece-wise linear recursion) is strictly more expressive than Datalog (with piece-wise linear recursion). This exposes one of the advantages of value invention that is available in Datalog$^{\exists}$-based languages, but not in plain Datalog.
(4) Finally, in Section 7 we provide initial evidence for the practical effect of piece-wise linearity on the Vadalog system, which has been originally designed for warded Datalog$^{\exists}$. In particular, we give preliminary experimental evidence for the effect of piece-wise linearity on the subsystem of Vadalog that is responsible for termination control, a crucial component that is needed due to the interaction of existential quantifiers and recursion.

As said above, this work is an extended and revised version of the conference paper [11]. Apart from giving the full proofs for all the results stated in [11], in addition, we include the following:

- A new alternating algorithm that allows to re-establish the complexity of ontological query answering under warded Datalog$^{\exists}$ in a transparent way.
- A discussion on the key model-theoretic property, known as unbounded ground-connection property, that distinguishes (piece-wise linear) warded Datalog$^{\exists}$ from existing formalisms.
- New query rewriting algorithms that allow us to rewrite (piece-wise linear) warded Datalog$^{\exists}$ queries into plain (piece-wise linear) Datalog queries.
- A preliminary experimental evaluation that provides evidence for the practical effect of piece-wise linearity on the Vadalog system.

2 PRELIMINARIES

Basics. We consider the disjoint countably infinite sets $\mathbf{C}$, $\mathbf{N}$, and $\mathbf{V}$ of *constants*, *(labeled) nulls*, and *variables*, respectively. The elements of $(\mathbf{C} \cup \mathbf{N} \cup \mathbf{V})$ are called *terms*. An *atom* is an expression of the form $R(\bar{t})$, where R is an n-ary predicate, and $\bar{t}$ is an n-tuple of terms. We write $\mathrm{var}(\alpha)$ for the set of variables in an atom α; this notation extends to sets of atoms. A *fact* is an atom that contains only constants. A *substitution* from a set of terms T to a set of terms T' is a function $h\colon T \to T'$. The restriction of h to a subset S of T, denoted $h_{|S}$, is the substitution $\{t \mapsto h(t) \mid t \in S\}$. A *homomorphism* from a set of atoms A to a set of atoms B is a substitution h from the set of terms in A to the set of terms in B such that h is the identity on $\mathbf{C}$, and $R(t_1, \ldots, t_n) \in A$ implies $h(R(t_1, \ldots, t_n)) = R(h(t_1), \ldots, h(t_n)) \in B$. We write $h(A)$ for the set of atoms $\{h(\alpha) \mid \alpha \in A\}$. For brevity, we may write $[n]$ for the set $\{1, \ldots, n\}$, where $n > 0$.

Relational Databases. A *schema* $\mathbf{S}$ is a finite set of relation symbols (or predicates), each having an associated *arity*. We write R/n to denote that R has arity $n \geq 0$. A *position* $R[i]$ in $\mathbf{S}$, where $R/n \in \mathbf{S}$ and $i \in [n]$, identifies the i-th argument of R. An *instance* over $\mathbf{S}$ is a (possibly infinite) set of atoms over $\mathbf{S}$ that contain constants and nulls, while a *database* over $\mathbf{S}$ is a finite set of facts over $\mathbf{S}$. The *active domain* of an instance I, denoted $\mathsf{dom}(I)$, is the set of all terms occurring in I.

Conjunctive Queries. A *conjunctive query* (CQ) over $\mathbf{S}$ is a first-order formula of the form

$$q(\bar{x}) \;\coloneqq\; \exists \bar{y}\, \big(R_1(\bar{z}_1) \wedge \cdots \wedge R_n(\bar{z}_n)\big),$$

where each $R_i(\bar{z}_i)$, for $i \in [n]$, is an atom without nulls over $\mathbf{S}$, each variable mentioned in the $\bar{z}_i$'s appears either in $\bar{x}$ or $\bar{y}$, and $\bar{x}$ are the *output variables* of q. For convenience, we adopt the rule-based syntax of CQs, i.e., a CQ as the one above will be written as the rule

$$Q(\bar{x}) \;\leftarrow\; R_1(\bar{z}_1), \ldots, R_n(\bar{z}_n),$$

where Q is a predicate used only in the head of CQs. Let $\mathsf{atoms}(q) = \{R_1(\bar{z}_1), \ldots, R_n(\bar{z}_n)\}$. The *evaluation* of $q(\bar{x})$ over an instance I, denoted $q(I)$, is the set of all tuples $h(\bar{x})$ of constants, where h is a homomorphism from $\mathsf{atoms}(q)$ to I.

Tuple-Generating Dependencies. A *tuple-generating dependency* (TGD) σ is a sentence

$$\forall \bar{x} \forall \bar{y}\, (\phi(\bar{x}, \bar{y}) \rightarrow \exists \bar{z}\, \psi(\bar{x}, \bar{z})),$$

where $\bar{x}, \bar{y}, \bar{z}$ are tuples of variables of $\mathbf{V}$, and ϕ, ψ are conjunctions of atoms without constants and nulls. For brevity, we write σ as $\phi(\bar{x}, \bar{y}) \rightarrow \exists \bar{z}\, \psi(\bar{x}, \bar{z})$, and use comma instead of $\wedge$ for joining atoms. We refer to ϕ and ψ as the *body* and *head* of σ, denoted $\mathsf{body}(\sigma)$ and $\mathsf{head}(\sigma)$, respectively. The *frontier* of the TGD σ, denoted $\mathsf{front}(\sigma)$, is the set of variables that appear both in the body and the head of σ. We also write $\mathrm{var}_{\exists}(\sigma)$ for the existentially quantified variables of σ. The schema of a set Σ of TGDs, denoted $\mathsf{sch}(\Sigma)$, is the set of predicates in Σ. An instance I satisfies a TGD σ as the one above, written $I \models \sigma$, if the following holds: whenever there exists a homomorphism h such

that $h(\phi(\bar{x}, \bar{y})) \subseteq I$, then there exists $h' \supseteq h_{|\bar{x}}$ such that $h'(\psi(\bar{x}, \bar{z})) \subseteq I$.[6] The instance I satisfies a set Σ of TGDs, written $I \models \Sigma$, if $I \models \sigma$ for each $\sigma \in \Sigma$.

Query Answering under TGDs. The main reasoning task under TGD-based languages is *conjunctive query answering*. Given a database D and a set Σ of TGDs, a *model* of D and Σ is an instance I such that $I \supseteq D$ and $I \models \Sigma$. Let $\mathrm{mods}(D, \Sigma)$ be the set of all models of D and Σ. The *certain answers* to a CQ q w.r.t. D and Σ is defined as the set tuples

$$\mathrm{cert}(q, D, \Sigma) \ := \bigcap_{I \in \mathrm{mods}(D,\Sigma)} q(I).$$

Our main task is to compute the certain answers to a CQ w.r.t. a database and a set of TGDs from a certain class C of TGDs; concrete classes of TGDs are discussed below. As is customary when studying the complexity of this problem, we focus on its decision version:

PROBLEM :	CQAns(C)
INPUT :	A database D, a set $\Sigma \in \mathsf{C}$ of TGDs, a CQ $q(\bar{x})$, and a tuple $\bar{c} \in \mathrm{dom}(D)^{\lvert\bar{x}\rvert}$.
QUESTION :	Is it the case that $\bar{c} \in \mathrm{cert}(q, D, \Sigma)$?

This general formulation refers to the *combined complexity* of the problem. However, we are also interested in the *data complexity*, which measures the complexity of the problem assuming that the set of TGDs $\Sigma \in \mathsf{C}$ and the CQ q are fixed, i.e., the complexity of the following problem:

PROBLEM :	$\mathrm{CQAns}(\Sigma, q(\bar{x}))$
INPUT :	A database D, and a tuple $\bar{c} \in \mathrm{dom}(D)^{\lvert\bar{x}\rvert}$.
QUESTION :	Is it the case that $\bar{c} \in \mathrm{cert}(q, D, \Sigma)$?

We adopt the usual convention and when we talk about the data complexity of CQAns(C) (i.e., the class of problems $\mathrm{CQAns}(\Sigma, q(\bar{x}))$, where Σ is a set of TGDs from C and q a CQ), we say that it is complete for a complexity class C if, for each set $\Sigma \in \mathsf{C}$ and CQ q, the problem $\mathrm{CQAns}(\Sigma, q(\bar{x}))$ is in C, and there exists a set of TGDs $\Sigma \in \mathsf{C}$ and a CQ q such that $\mathrm{CQAns}(\Sigma, q(\bar{x}))$ is C-hard.

The Chase Procedure. A useful algorithmic tool for tackling the above problem is the well-known *chase procedure*; see, e.g., [14, 16, 19, 22]. We start by defining a single chase step. Consider a set Σ of TGDs and an instance I. A *trigger* for Σ on I is a pair (σ, h), where $\sigma \in \Sigma$ and h is a homomorphism from $\mathrm{body}(\sigma)$ to I. An *application* of (σ, h) to I returns the instance $J = I \cup \{h'(\psi(\bar{x}, \bar{z}))\}$, where $h' \supseteq h_{\mathrm{front}(\sigma)}$ is such that (i) for each existentially quantified variable z of σ, $h'(z) \in \mathbf{N} \setminus \mathrm{dom}(I)$, i.e., $h'(z)$ is a "fresh" null not occurring in I, and (ii) for each pair (z, w) of distinct existentially quantified variables of σ, $h'(z) \neq h'(w)$. Such a single chase step is denoted $I\langle\sigma, h\rangle J$.

The main idea of the chase is, starting from a database D, to exhaustively apply triggers for the given set Σ of TGDs on the instance constructed so far. Formally, a *chase sequence for D under Σ* is a (possibly infinite) sequence of instances $\delta = I_0, I_1, I_2, \ldots$, with $I_0 = D$, such that

(1) for each $i \geq 0$, there exists a trigger (σ, h) for Σ on I_i such that $I_i\langle\sigma, h\rangle I_{i+1}$, i.e., I_{i+1} is obtained via an application of a trigger to I_i,
(2) for each $i \geq 0$ and $j > i$, assuming that $I_i\langle\sigma_i, h_i\rangle I_{i+1}$ and $I_j\langle\sigma_j, h_j\rangle I_{j+1}$, the following holds: $\sigma_i = \sigma_j$ implies $h_i \neq h_j$, i.e., we cannot use a trigger that has been already applied, and
(3) for each $i \geq 0$, and for every trigger (σ, h) for Σ on I_i, there exists $j \geq i$ such that $I_j\langle\sigma, h\rangle I_{j+1}$; this is known as the *fairness condition*, which guarantees that all the triggers will be applied.

[6] By abuse of notation, we may treat a tuple of variables as a set of variables, and a conjunction of atoms as a set of atoms.

The *result of* δ is defined as the instance $\bigcup_{i \geq 0} I_i$, which always exists. It is easy to verify that there are several chase sequences for D under Σ depending on the order in which we apply the TGDs. Nevertheless, the fact that we consider the oblivious version of the chase, i.e., we blindly apply a trigger without checking whether the head is already satisfied, together with the fairness condition, allow us to show that different chase sequences for D under Σ lead to the same result (up to isomorphism). Hence, we can refer to *the* result of the chase for D under Σ, denoted $\mathsf{chase}(D, \Sigma)$. The following is a classical result that collects some useful properties of the chase:

Proposition 2.1. *Consider a database D, and a set Σ of TGDs. The following hold:*

(1) $\mathsf{chase}(D, \Sigma) \in \mathsf{mods}(D, \Sigma)$.

(2) For every $I \in \mathsf{mods}(D, \Sigma)$*, there exists a homomorphism from* $\mathsf{chase}(D, \Sigma)$ *to* I.

3 THE LOGICAL CORE OF VADALOG

A crucial component of the Vadalog system is its reasoning engine, which in turn is built around the Vadalog language, a general-purpose formalism for knowledge representation and reasoning. The logical core of this language is the well-behaved class of warded sets of TGDs [3, 18].

3.1 An Intuitive Description

Wardedness applies a syntactic restriction on how certain "dangerous" variables of a set of TGDs are used. These are body variables that can be unified with a null during the chase, and that are also propagated to the head. For example, given the TGDs

$$P(x) \rightarrow \exists z\, R(x, z) \qquad \text{and} \qquad R(x, y) \rightarrow P(y)$$

the variable y in the body of the second TGD is dangerous. Indeed, once the chase applies the first TGD, an atom of the form $R(\_, \bot)$ is generated, where $\bot$ is a null value, and then the second TGD is triggered with the variable y being unified with $\bot$ that is propagated to the obtained atom $P(\bot)$. It has been observed in the TGD literature that the liberal use of dangerous variables leads to a prohibitively high computational complexity of the main reasoning tasks, in particular of CQ answering [14]. The main goal of wardedness is to limit the use of dangerous variables with the aim of taming the way that null values are propagated during the execution of the chase procedure. This is achieved by posing the following two conditions:

(1) the dangerous variables should appear together in a single body atom α, called a ward, and
(2) the atom α can share only harmless variables with the rest of the body, that is, variables that unify only with constants.

We proceed to formalize the above description.

3.2 The Formal Definition

We first need some auxiliary notions. The set of positions of a schema $\mathbf{S}$, denoted $\mathsf{pos}(\mathbf{S})$, is defined as $\{R[i] \mid R/n \in \mathbf{S}, \text{ with } n \geq 1, \text{ and } i \in [n]\}$. Given a set Σ of TGDs, we write $\mathsf{pos}(\Sigma)$ instead of $\mathsf{pos}(\mathsf{sch}(\Sigma))$. The set of *affected positions* of $\mathsf{sch}(\Sigma)$, denoted $\mathsf{aff}(\Sigma)$, is inductively defined as follows:

- if there exists $\sigma \in \Sigma$ and a variable $x \in \mathsf{var}_{\exists}(\sigma)$ at position π, then $\pi \in \mathsf{aff}(\Sigma)$, and
- if there exists $\sigma \in \Sigma$ and a variable $x \in \mathsf{front}(\sigma)$ occurring in the body of σ only at positions of $\mathsf{aff}(\Sigma)$, and x appears in the head of σ at position π, then $\pi \in \mathsf{aff}(\Sigma)$.

Let $\mathsf{nonaff}(\Sigma) = \mathsf{pos}(\Sigma) \setminus \mathsf{aff}(\Sigma)$. We can now classify the variables in the body of a TGD into harmless, harmful, and dangerous. Fix a TGD $\sigma \in \Sigma$ and a variable x in $\mathsf{body}(\sigma)$:

- x is *harmless* if at least one occurrence of it appears in $\mathsf{body}(\sigma)$ at a position of $\mathsf{nonaff}(\Sigma)$,
- x is *harmful* if it is not harmless, and

- x is *dangerous* if it is harmful and belongs to front(σ).

We are now ready to formally introduce wardedness.

*Definition 3.1 (**Wardedness**).* A set Σ of TGDs is *warded* if, for each $\sigma \in \Sigma$, there are no dangerous variables in body(σ), or there exists an atom $\alpha \in$ body(σ), called a *ward*, such that:

(1) all the dangerous variables in body(σ) occur in α, and
(2) each variable of var(α) $\cap$ var(body(σ) $\setminus \{\alpha\}$) is harmless.

We write WARD for the class of all finite warded sets of TGDs. ∎

Let us clarify that warded sets of TGDs form a subclass of a highly expressive class of TGDs known in the literature as weakly-frontier-guarded sets of TGDs [6]. In fact, the only difference between those two classes is the second condition given in Definition 3.1. In other words, if we drop the second condition in the definition of wardeness, then we get the class of weakly-frontier-guarded sets of TGDs. However, as shown in [5], query answering under weakly-frontier-guarded sets of TGDs is prohibitively complex, namely ExpTime-complete in data complexity, whereas, as recently shown in [3, 18], query answering under warded sets of TGDs is tractable in data complexity. In fact, as thoroughly discussed in [3], wardedness forms a "nearly" maximal tractable fragment of weakly-frontier-guarded sets of TGDs.

Proposition 3.2. CQAns(WARD) *is* ExpTime*-complete in combined complexity, and* PTime*-complete in data complexity.*

Let us clarify that [3, 18] deal only with the data complexity of the problem CQAns(WARD). However, the same algorithm provides an ExpTime upper bound in combined complexity. The lower bounds are inherited from Datalog since a set of Datalog rules (seen as TGDs) is warded.

3.3 Why is Wardedness Useful?

One of the distinctive features of wardedness, which is crucial for the purposes of the Vadalog system, is the fact that it can express every SPARQL query under the OWL 2 QL direct semantics entailment regime, which is inherited from the OWL 2 direct semantics entailment regime; for details, see [3, 18, 21]. Recall that SPARQL is the standard language for querying the Semantic Web,[7] while OWL 2 QL is a prominent profile of OWL 2.[8] We give a very simple example of a warded set of TGDs, which is extracted from the set of TGDs that encodes the OWL 2 direct semantics entailment regime for OWL 2 QL. Actually, for the sake of readability, we give a simplified version of this set, while the proper definition can be found in [3].

Example 3.3. An OWL 2 QL ontology can be stored in a database using atoms of the form Restriction(c, p) stating that the class c is a restriction of the property p, SubClass(c, c') stating that c is a subclass of c', and Inverse(p, p') stating that p is the inverse property of p'. We can then compute all the logical inferences of the given ontology using TGDs as the ones below:

$$\underline{\text{Type}(x, y)}, \text{SubClass}(y, z) \rightarrow \text{Type}(x, z)$$
$$\underline{\text{Type}(x, y)}, \text{Restriction}(y, z) \rightarrow \exists w\, \text{Triple}(x, z, w)$$
$$\underline{\text{Triple}(x, y, z)}, \text{Inverse}(y, w) \rightarrow \text{Triple}(z, w, x)$$
$$\underline{\text{Triple}(x, y, z)}, \text{Restriction}(w, y) \rightarrow \text{Type}(x, w).$$

The first TGD transfers the class type, i.e., if a is of type b and b is a subclass of c, then a is also of type c. Moreover, the second TGD states that if a is of type b and b is the restriction of the property

[7] http://www.w3.org/TR/rdf-sparql-query
[8] https://www.w3.org/TR/owl2-overview/

p, then a is related to some c via the property p, which is encoded by the atom $\text{Triple}(a, p, c)$. Analogously, the last two TGDs encode the usual meaning of inverses and the effect of restrictions on types. It is easy to verify that the above set of TGDs is warded, where the underlined atoms are the wards. A variable in an atom with predicate Restriction, SubClass, or Inverse, is trivially harmless. The frontier variables that appear at Type[1], Triple[1], or Triple[3], are dangerous, and the underlined atoms are acting as wards. ∎

Although there are several tractable fragments of weakly-frontier-guarded sets of TGDs other than warded sets of TGDs (e.g., frontier-guarded TGDs [6]), it turned out that these are not powerful enough for expressing every SPARQL query under the OWL 2 QL direct semantics entailment regime, or even the simple set of TGDs given in Example 3.3 [3]. The proof of this result relies on a key model-theoretic property that demonstrates the difference between wardedness and the other tractable classes of weakly-frontier-guarded sets of TGDs that can be found in the literature. We proceed to recall this property, which, of course, should be preserved when we are going to limit the recursion allowed by wardedness towards the space-efficient core of Vadalog.

Roughly speaking, a class C of TGDs has the so-called unbounded ground-connection property if it allows us to connect in the chase, via a fixed set of TGDs, an invented null value with an unbounded number of constants occurring in the input database. Given an instance I, the *ground connection* of a null $\bot \in (\text{dom}(I) \cap \mathbf{N})$ in I, denoted $\text{gc}(\bot, I)$, is defined as the set of constants

$$\{c \in \mathbf{C} \mid \text{ there exists } R(t_1, \ldots, t_n) \in I \text{ such that } \{c, \bot\} \subseteq \{t_1, \ldots, t_n\}\},$$

i.e., all the constants that jointly appear with $\bot$ in an atom of I. For a set Σ of TGDs, and a family of databases $(D_n)_{n>0}$, we define the function

$$\text{mgc}(n) = \begin{cases} \max_{\bot \in \text{dom}(\text{chase}(D_n,\Sigma)) \cap \mathbf{N}} \{|\text{gc}(\bot, \text{chase}(D_n, \Sigma))|\} & \text{if } (\text{dom}(\text{chase}(D_n, \Sigma)) \cap \mathbf{N}) \neq \emptyset, \\ 0 & \text{otherwise.} \end{cases}$$

A class C of TGDs has the *unbounded ground-connection property* (UGCP) if there exists a set of TGDs $\Sigma \in \mathbf{C}$, and a family of databases $(D_n)_{n>0}$, such that $\text{mgc}(n) \notin O(1)$.

By exploiting the warded set of TGDs given in Example 3.3, we can show that WARD enjoys the UGCP, a result that is implicit in [3]. Consider the family of databases $(D_n)_{n>0}$, where

$$D_n = \{\text{Inverse}(p, p^-), \text{Restriction}(c_p, p), \text{Restriction}(c_p^-, p^-), \text{Type}(a, c_0), \text{SubClass}(c_0, c_p) \\ \text{SubClass}(c_p^-, c_1), \text{SubClass}(c_1, c_2), \text{SubClass}(c_2, c_3), \ldots, \text{SubClass}(c_{n-1}, c_n)\}.$$

It is easy to verify that the instance $\text{chase}(D_n, \Sigma)$, where Σ is the set of TGDs given in Example 3.3, contains (among others) the atoms

$$\text{Type}(\bot, c_1), \text{Type}(\bot, c_2), \ldots, \text{Type}(\bot, c_n),$$

where $\bot$ is a null. Therefore, the ground connection of $\bot$ in $\text{chase}(D_n, \Sigma)$ contains the constants $c_1, \ldots, c_n$. Thus, $\text{mgc}(n) \notin O(1)$, which in turn implies the desired result:

Proposition 3.4. WARD *has the unbounded ground-connection property.*

4 LIMITING RECURSION

We now focus on our main research question: can we limit the recursion allowed by wardedness in order to obtain a formalism that provides a convenient syntax for expressing useful recursive statements, and at the same time achieves space-efficiency? The above question has been extensively studied in the 1980s for Datalog programs, with *linear Datalog* being a key fragment that achieves a good balance between expressivity and complexity; see, e.g., [23, 24]. A Datalog program Σ is *linear*

if, for each rule in Σ, its body contains at most one atom that uses an intensional predicate, i.e., a predicate that appears in the head of at least one rule of Σ. In other words, linear Datalog allows only for linear recursion, which is able to express many real-life recursive queries. However, for our purposes, linear recursion does not provide the convenient syntax that we are aiming at. After analyzing several real-life examples of warded sets of TGDs, provided by our industrial partners, we observed that the employed recursion goes beyond linear recursion. On the other hand, most of the examples coming from our industrial partners use recursion in a restrictive way: each TGD has at most one body atom whose predicate is mutually recursive with a predicate occurring in the head of the TGD. Interestingly, this more liberal version of linear recursion has been already investigated in the context of Datalog, and it is known as *piece-wise linear*; see, e.g., [1]. Does this type of recursion lead to the space-efficient fragment of warded sets of TGDs that we are looking for? The rest of this section is devoted to providing an affirmative answer to this question.

4.1 Piece-Wise Linearity

Let us start by formally defining the class of piece-wise linear sets of TGDs. To this end, we need to define when two predicates are mutually recursive, which in turn relies on the well-known notion of the predicate graph. The *predicate graph* of a set Σ of TGDs, denoted $\mathsf{pg}(\Sigma)$, is a directed graph (V, E), where $V = \mathsf{sch}(\Sigma)$, and there exists an edge from a predicate P to a predicate R, i.e., $(P, R) \in E$, iff there exists a TGD $\sigma \in \Sigma$ such that P occurs in $\mathsf{body}(\sigma)$ and R occurs in $\mathsf{head}(\sigma)$. Two predicates $P, R \in \mathsf{sch}(\Sigma)$ are *mutually recursive* (w.r.t. Σ) if there exists a cycle in $\mathsf{pg}(\Sigma)$ that contains both P and R (i.e., R is reachable from P, and vice versa).

*Definition 4.1 (**Piece-Wise Linearity**).* A set Σ of TGDs is *piece-wise linear* if, for each TGD $\sigma \in \Sigma$, there exists at most one atom in $\mathsf{body}(\sigma)$ whose predicate is mutually recursive with a predicate in $\mathsf{head}(\sigma)$. We write PWL for the class of all finite piece-wise linear sets of TGDs. ∎

The main result of this section follows:

Theorem 4.2. CQAns(WARD∩PWL) *is* PSpace*-complete in combined complexity, and* NLogSpace*-complete in data complexity.*

The lower bounds are inherited from linear Datalog. The difficult task is to establish the upper bounds. This relies on a novel notion of proof tree that is of independent interest. As we shall see, our notion of proof tree leads to space-bounded algorithms that allow us to show the upper bounds in Theorem 4.2, and also re-establish in a transparent way the upper bounds in Proposition 3.2. Moreover, in Section 6, we are going to use our novel notion of proof tree for studying the relative expressive power of the query languages based on (piece-wise linear) warded sets of TGDs.

Before we proceed with the notion of proof tree, let us stress that piece-wise linearity preserves the unbounded ground-connection property discussed in the previous section. In fact, this is shown in exactly the same way as Proposition 3.4, by relying on the warded set of TGDs given in Example 3.3, which is also piece-wise linear. This essentially tells us that even if we restrict wardedness by allowing only piece-wise linear recursion, the obtained class is able to express useful statements, for example, encoding the OWL 2 QL direct semantics entailment regime, that are provably not expressible by existing tractable fragments of weakly-frontier-guarded sets of TGDs.

Proposition 4.3. WARD ∩ PWL *has the unbounded ground-connection property.*

4.2 Query Answering via Proof Trees

It is known that given a CQ q and a set Σ of TGDs, we can unfold q using the TGDs of Σ into an infinite union of CQs q_Σ such that, for every database D, $\mathsf{cert}(q, D, \Sigma) = q_\Sigma(D)$; see, e.g., [17, 20]. Let

us clarify that in our context, an unfolding, which is essentially a resolution step, is more complex than in the context of Datalog due to the existentially quantified variables in the head of TGDs. The intention underlying our notion of proof tree is to encode in a tree the sequence of CQs, generated during the unfolding of q with Σ, that leads to some CQ q' of q_Σ. In particular, each intermediate CQ, as well as q', is carefully decomposed into smaller subqueries that form the nodes of the tree with q being the root, and q' being the CQ obtained after merging the leaves. As we shall see, if we focus on well-behaved classes of TGDs such as (piece-wise linear) warded sets of TGDs, we can establish upper bounds on the size of those subqueries, which in turn allow us to devise space-bounded algorithms for query answering. In what follows, we define the notion of proof tree (Definition 4.8), and establish its correspondence with query answering (Theorem 4.9). To this end, we need to introduce the main building blocks of a proof tree: chunk-based resolution (Definition 4.4), a query decomposition technique (Definition 4.6), and the notion of specialization for CQs (Definition 4.7).

Chunk-based Resolution. Let A and B be non-empty sets of atoms that mention only constants and variables. The sets A and B *unify* if there is a substitution γ, which is the identity on $\mathbf{C}$, called *unifier for A and B*, such that $\gamma(A) = \gamma(B)$. A *most general unifier* (MGU) for A and B is a unifier for A and B, denoted $\gamma_{A,B}$, such that, for each unifier γ for A and B, $\gamma = \gamma' \circ \gamma_{A,B}$ for some substitution γ'. It is well-known that if two sets of atoms unify, then there is always a MGU, which is unique (modulo variable renaming). Given a CQ $q(\bar{x})$ and a set of atoms $S \subseteq \mathsf{atoms}(q)$, we say that a variable $y \in \mathsf{var}(S)$ is *shared*, if $y \in \bar{x}$ or $y \in \mathsf{var}(\mathsf{atoms}(q) \setminus S)$. A *chunk unifier* of q with a TGD σ (where q and σ do not share variables) is a triple (S_1, S_2, γ), where $\emptyset \subset S_1 \subseteq \mathsf{atoms}(q)$, $\emptyset \subset S_2 \subseteq \mathsf{head}(\sigma)$, and γ is a unifier for S_1 and S_2 such that, for each $x \in \mathsf{var}(S_2) \cap \mathsf{var}_\exists(\sigma)$,

(1) $\gamma(x) \notin \mathbf{C}$, i.e., $\gamma(x)$ is not constant, and
(2) for every variable y different from x, $\gamma(x) = \gamma(y)$ implies y occurs in S_1 and is not shared.

The chunk unifier (S_1, S_2, γ) is *most general* (MGCU) if γ is an MGU for S_1 and S_2. Notice that the variables of $\mathsf{var}_\exists(\sigma)$ occurring in S_2 unify (via γ) only with non-shared variables of S_1. This ensures that S_1 is a "chunk" of q that can be resolved as a whole via σ using γ.[9] Consider, for example,

$$q \;=\; Q(x) \leftarrow R(x,y), S(y) \quad \text{and} \quad \sigma \;=\; P(x') \rightarrow \exists y'\, R(x',y'), S(y').$$

It should be clear that $R(x,y), S(y)$ is a "chunk" of q that can be resolved with σ using $\gamma = \{x \mapsto x', y \mapsto y', x' \mapsto x', y' \mapsto y'\}$; in fact, using the MGCU $(\mathsf{atoms}(q), \mathsf{head}(\sigma), \gamma)$. On the other hand, the atom $R(x,y)$ alone should not be considered as a "chunk" of q that can be resolved with σ using γ, even though γ is a unifier for $\{R(x,y)\}$ and $\{R(x',y')\}$. Such a resolution step would be unsound since the shared variable y in $R(x,y)$ is lost due to its unification with the existentially quantified variable y'. Thus, the triple $(\{R(x,y)\}, \{R(x',y')\}, \gamma)$ should not be a valid chunk unifier of q with σ, which is guaranteed by the additional conditions on γ in the definition of chunk unifier.

*Definition 4.4 (**Chunk-based Resolution**).* Let $q(\bar{x})$ be a CQ and σ a TGD. A *σ-resolvent* of q is a CQ $q'(\gamma(\bar{x}))$ with $\mathsf{atoms}(q') = \gamma((\mathsf{atoms}(q) \setminus S_1) \cup \mathsf{body}(\sigma))$ for a MGCU (S_1, S_2, γ) of q with σ. ■

As discussed above, the purpose of a proof tree is to encode a finite branch of the unfolding of a CQ q with a set Σ of TGDs, which is obtained by applying chunk-based resolution. Such a branch is a sequence $q_0, \ldots, q_n$ of CQs, where $q = q_0$, and, for each $i \in [n]$, q_i is a σ-resolvent of q_{i-1} for some $\sigma \in \Sigma$. Here is a simple example, which will serve as a running example in the rest of the section, that illustrates the notion of unfolding.

[9] A similar notion known as piece unifier has been defined in [20]. We adopted the term chunk unifier in order to avoid confusion with the term piece-wise linear.

Example 4.5. Consider the set Σ of TGDs consisting of

$$R(x) \rightarrow \exists y\, T(y,x) \qquad T(x,y), S(y,z) \rightarrow T(x,z) \qquad T(x,y), P(y) \rightarrow G()$$

and the CQ that simply asks whether $G()$ is entailed, i.e., the CQ

$$Q \leftarrow G().$$

Since the unfolding of q with Σ should give the correct answer for *every* input database, and thus, for databases of the form $\{R(c^{n-1}), S(c^{n-1}, c^{n-2}), \ldots, S(c^2, c^1), P(c^1)\}$, for some $n > 1$, one of its branches should be $q = q_0, q_1, \ldots, q_n$, where

$$q_1 = Q \leftarrow T(x, y^1), P(y^1)$$

obtained by resolving q_0 using the third TGD,

$$q_i = Q \leftarrow T(x, y^i), S(y^i, y^{i-1}), \ldots, S(y^2, y^1), P(y^1),$$

for $i \in \{2, \ldots, n-1\}$, obtained by resolving q_{i-1} using the second TGD, and finally

$$q_n = Q \leftarrow R(y^{n-1}), S(y^{n-1}, y^{n-2}), \ldots, S(y^2, y^1), P(y^1)$$

obtained by resolving q_{n-1} using the first TGD. ∎

At this point, one may think that the proof tree that encodes the branch $q_0, \ldots, q_n$ of the unfolding of q with Σ is the finite labeled path $v_0, \ldots, v_n$, where each v_i is labeled by q_i. However, another crucial goal of such a proof tree, which is not achieved via the naive path encoding, is to split each resolvent q_i, for $i > 0$, into smaller subqueries $q_i^1, \ldots, q_i^{n_i}$, which are essentially the children of q_i, in such a way that they can be processed independently by resolution. The crux of this encoding is that it provides us with a mechanism for keeping the CQs that must be processed by resolution small. It should be clear from Example 4.5 that by following the naive path encoding, without splitting the resolvents into smaller subqueries, we may get CQs of unbounded size. This brings us to the notion of query decomposition.

Query Decomposition. The key question here is how a CQ q can be decomposed into subqueries that can be processed independently. The subtlety is that, after splitting q, occurrences of the same variable may be separated into different subqueries. Therefore, we need a way to ensure that a variable in q, which appears in different subqueries after the splitting, is indeed treated as the same variable, i.e., it has the same meaning. We deal with this issue by restricting the set of variables in q of which occurrences can be separated during the splitting step. In particular, we can only separate occurrences of an output variable. This relies on the convention that output variables correspond to fixed constant values of $\mathbf{C}$, and thus their name is "frozen" and never renamed by subsequent resolution steps. Hence, we can separate occurrences of an output variable into different subqueries, i.e., different branches of the proof tree, without losing the semantic connection between them. Summing up, the main idea underlying query decomposition is to split the CQ at hand into smaller queries that keep together all the occurrences of a non-output variable, but with the freedom of separating occurrences of an output variable. The formal definition follows.

*Definition 4.6 (**Query Decomposition**).* Given a CQ $q(\bar{x})$, a *decomposition* of q is a set of CQs $\{q_1(\bar{y}_1), \ldots, q_n(\bar{y}_n)\}$, where $n \geq 1$ and $\bigcup_{i \in [n]} \mathsf{atoms}(q_i) = \mathsf{atoms}(q)$, such that, for each $i \in [n]$:

(1) $\bar{y}_i$ is the restriction of $\bar{x}$ on the variables in q_i, and
(2) for every $\alpha, \beta \in \mathsf{atoms}(q)$, if $\alpha \in \mathsf{atoms}(q_i)$ and $\mathsf{var}(\alpha) \cap \mathsf{var}(\beta) \not\subseteq \bar{x}$, then $\beta \in \mathsf{atoms}(q_i)$. ∎

An example that stresses the usefulness of query decomposition combined with the notion of query specialization is given below.

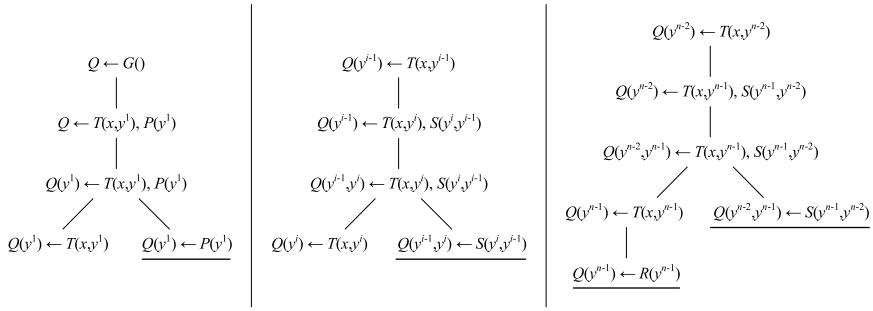


Fig. 1. Proof tree that encodes the branch $Q = q_0, \ldots, q_n$ of the unfolding of q with Σ from Example 4.5.

Query Specialization. From the above discussion, one expects that a proof tree of a CQ q w.r.t. a set Σ of TGDs can be constructed by starting from q, which is the root, and applying two steps: resolution and decomposition. However, this is not enough for our purposes as we may run into the following problem: some of the subqueries will mistakenly remain large since we have no way to realize that a non-output variable corresponds to a fixed constant value, which in turn allows us to "freeze" its name and separate different occurrences of it during the decomposition step. This is illustrated by Example 4.5. Observe that the size of the CQs $\{q_i\}_{i>0}$ grows arbitrarily, while our query decomposition has no effect on them since they are Boolean queries, i.e., queries without output variables, and thus, we cannot split them into smaller subqueries. This issue can be solved via an intermediate step between resolution and decomposition, the so-called specialization step. A specialization of a CQ is obtained by converting some of the non-output variables into output variables, while keeping their name, or taking the name of an existing output variable.

*Definition 4.7 (**Query Specialization**).* Let $q(\bar{x})$ be a CQ with $\text{atoms}(q) = \{\alpha_1, \ldots, \alpha_n\}$. A *specialization* of q is a CQ $Q(\bar{x}, \bar{y}) \leftarrow \rho_{\bar{z}}(\alpha_1, \ldots, \alpha_n)$, where $\bar{y}, \bar{z}$ are (possibly empty) disjoint tuples of non-output variables of q, and $\rho_{\bar{z}}$ is a substitution from $\bar{z}$ to $\bar{x} \cup \bar{y}$. ∎

Consider, for example, the CQ q_1 from Example 4.5

$$Q \leftarrow T(x, y^1), P(y^1)$$

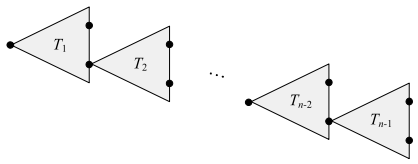
obtained by resolving $q = q_0$ using the third TGD. The query decomposition cannot split this query into smaller subqueries since the variable y^1 is a non-output variable, and thus, all its occurrences should be kept together. We can consider the following specialization of q_1

$$Q(y^1) \leftarrow T(x, y^1), P(y^1),$$

which converts y^1 into an output variable, and now by query decomposition we can split it into

$$Q(y^1) \leftarrow T(x, y^1) \qquad Q(y^1) \leftarrow P(y^1).$$

Proof Trees. We are now ready to introduce our new notion of proof tree. We first explain the high-level idea by exploiting our running example. Consider the set Σ of TGDs and the CQ q from Example 4.5. The branch $q_0, \ldots, q_n$ of the unfolding of q with Σ given in Example 4.5 is encoded via a proof tree of the form



where each T_i, for $i \in [n-1]$, is a rooted tree with only two leaf nodes. The actual trees are depicted in Figure 1; the left one is T_1, the middle one is T_i for $i \in \{2, \ldots, n-2\}$, while the right one is T_{n-1}. For each $i \in [n-1]$, the child of the root of T_i is obtained via resolution, then we specialize it by converting the variable y^i into an output variable, and then we decompose the specialized CQ into two subqueries. In T_{n-1}, we also apply an additional resolution step in order to obtain the leaf node $Q(y^{n-1}) \leftarrow R(y^{n-1})$. The underlined CQs are the subqueries that correspond to q_n of the unfolding. Indeed, the conjunction of the atoms occurring in the underlined CQs is precisely the CQ q_n.

We proceed to give the formal definition. Given a partition $\pi = \{S_1, \ldots, S_m\}$ of a set of variables, we write eq_π for the substitution that maps the variables of S_i to the same variable x_i, where x_i is a distinguished element of S_i. We should not forget the convention that output variables cannot be renamed, and thus, a resolution step should use a MGCU that preserves the output variables. In particular, given a CQ q and a TGD σ, a σ-resolvent of q is called *IDO* if the underlying MGCU uses a substitution that is the identity on the output variables of q (hence the name IDO). Finally, given a TGD σ and some object o (e.g., o can be the node of a tree, or an integer number), we write σ_o for the TGD obtained by renaming each variable x in σ into a new variable x_o not occurring in σ. This is a simple mechanism that allows us to uniformly rename the variables of a TGD in order to avoid undesirable clatter among variables during a resolution step.

*Definition 4.8 (**Proof Tree**).* Let $q(\bar{x})$ be a CQ with $\mathrm{atoms}(q) = \{\alpha_1, \ldots, \alpha_n\}$, and Σ a set of TGDs. A *proof tree* of q w.r.t. Σ is a triple $\mathcal{P} = (T, \lambda, \pi)$, where $T = (V, E)$ is a finite rooted tree, λ a labeling function that assigns a CQ to each node of T, and π a partition of $\bar{x}$, such that, for each $v \in V$:

(1) If v is the root node of T, then $\lambda(v)$ is the CQ $Q(\mathrm{eq}_\pi(\bar{x})) \leftarrow \mathrm{eq}_\pi(\alpha_1, \ldots, \alpha_n)$.
(2) If v has only one child u, then:
  (a) there exists $\sigma \in \Sigma$ such that $\lambda(u)$ is an IDO σ_v-resolvent of $\lambda(v)$, or
  (b) $\lambda(u)$ is a specialization of $\lambda(v)$.
(3) If v has the children $u_1, \ldots, u_k$ for $k > 1$, then $\{\lambda(u_1), \ldots, \lambda(u_k)\}$ is a decomposition of $\lambda(v)$.

Assuming that $v_1, \ldots, v_m$ are the leaf nodes of T, the *CQ induced by* $\mathcal{P}$ is defined as

$$Q(\mathrm{eq}_\pi(\bar{x})) \leftarrow \alpha_1, \ldots, \alpha_\ell,$$

where $\{\alpha_1, \ldots, \alpha_\ell\} = \bigcup_{i \in [m]} \mathrm{atoms}(\lambda(v_i))$. ∎

Note that the purpose of the partition π is to indicate that some output variables correspond to the same constant value – this is why variables in the same set of π are unified via the substitution eq_π. This unification step is crucial in order to safely use, in subsequent resolution steps, substitutions that are the identity on the output variables. If we omit this initial unification step, we may lose important resolution steps, and thus being incomplete for query answering purposes.

The main result of this section, which exposes the connection between proof trees and CQ answering, follows. By abuse of notation, we write $\mathcal{P}$ for the CQ induced by $\mathcal{P}$.

Theorem 4.9. *Consider a database D, a set Σ of TGDs, a CQ $q(\bar{x})$, and a tuple $\bar{c} \in \mathrm{dom}(D)^{|\bar{x}|}$. The following are equivalent:*

(1) $\bar{c} \in \mathrm{cert}(q, D, \Sigma)$.
(2) There is a proof tree $\mathcal{P}$ of q w.r.t. Σ such that $\bar{c} \in \mathcal{P}(D)$.

The proof of the above result, which can be found in the appendix, relies on the soundness and completeness of chunk-based resolution. Given a set Σ of TGDs and a CQ $q(\bar{x})$, by exhaustively applying chunk-based resolution, we can construct a (possibly infinite) union of CQs q_Σ such that, for every database D, $\mathsf{cert}(q, D, \Sigma) = q_\Sigma(D)$. In other words, given a tuple $\bar{c} \in \mathsf{dom}(D)^{|\bar{x}|}$, $\bar{c} \in \mathsf{cert}(q, D, \Sigma)$ iff there exists a CQ $q'(\bar{x})$ in q_Σ such that $\bar{c} \in q'(D)$. We can then show that the latter statement is equivalent to the existence of a proof tree $\mathcal{P}$ of q w.r.t. Σ such that $\bar{c} \in \mathcal{P}(D)$.

4.3 Well-behaved Proof Trees

Theorem 4.9 states that checking whether a tuple $\bar{c}$ is a certain answer boils down to deciding whether there exists a proof tree $\mathcal{P}$ such that $\bar{c}$ is an answer to the CQ induced by $\mathcal{P}$ over the given database. Of course, in general, the latter is an undecidable problem. However, if we focus on (piece-wise linear) warded sets of TGDs, it suffices to check for the existence of a well-behaved proof tree with certain syntactic properties, which in turn allows us to devise a decision procedure. We proceed to make this statement more precise.

For clarity, we focus on *single-head* TGDs, i.e., TGDs with only one atom in the head, since we can always normalize a warded set of TGDs into single-head TGDs, while the certain answers are preserved. This relies on the following simple transformation. Given a TGD σ of the form

$$\phi(\bar{x}, \bar{y}) \rightarrow \exists \bar{z}\, R_1(\bar{x}_1, \bar{z}_1), \ldots, R_n(\bar{x}_n, \bar{z}_n),$$

where $\bar{x}_i \subseteq \bar{x}$ and $\bar{z}_i \subseteq \bar{z}$ for each $i \in [n]$, let $\mathsf{S}(\sigma)$ be the set of single-head TGDs consisting of

$$\begin{aligned} \phi(\bar{x}, \bar{y}) &\rightarrow \exists \bar{z}\, \mathrm{Aux}_\sigma(\bar{x}, \bar{z}) \\ \mathrm{Aux}_\sigma(\bar{x}, \bar{z}) &\rightarrow R_1(\bar{x}_1, \bar{z}_1) \\ &\vdots \\ \mathrm{Aux}_\sigma(\bar{x}, \bar{z}) &\rightarrow R_n(\bar{x}_n, \bar{z}_n). \end{aligned}$$

We then define $\mathsf{N}_{\mathsf{sh}}(\Sigma)$, that is, the normalization of Σ into single-head TGDs, as the set $\bigcup_{\sigma \in \Sigma} \mathsf{S}(\sigma)$. The next easy lemma collects some useful facts about the normalization procedure $\mathsf{N}_{\mathsf{sh}}(\cdot)$:

Lemma 4.10. *Consider a set Σ of TGDs. The following hold:*

(1) $\mathsf{N}_{\mathsf{sh}}(\Sigma)$ can be computed in polynomial time in the size of Σ.
(2) For $\mathsf{C} \in \{\mathsf{WARD} \cap \mathsf{PWL}, \mathsf{WARD}\}$, $\Sigma \in \mathsf{C}$ implies $\mathsf{N}_{\mathsf{sh}}(\Sigma) \in \mathsf{C}$.
(3) Given a database D, and a CQ $q(\bar{x})$, $\mathsf{cert}(q, D, \Sigma) = \mathsf{cert}(q, D, \mathsf{N}_{\mathsf{sh}}(\Sigma))$.[10]

Piece-wise Linear Warded Sets of TGDs. For piece-wise linear warded sets of TGDs, we can strengthen Theorem 4.9 by focussing on proof trees that enjoy two syntactic properties:

(1) they have a path-like structure, and
(2) the size of the CQs that label their nodes is bounded by a polynomial.

The first property is formalized via linear proof trees. Let $\mathcal{P} = (T, \lambda, \pi)$, where $T = (V, E)$, be a proof tree of a CQ q w.r.t. a set Σ of TGDs. We call $\mathcal{P}$ *linear* if, for each node $v \in V$, there exists at most one node $u \in V$ such that $(v, u) \in E$ and u is not a leaf in T, i.e., v has at most one child that is not a leaf. For example, the proof tree given above, which consists of the trees depicted in Figure 1, is linear. The second property relies on the notion of *node-width* of $\mathcal{P}$ defined as

$$\mathsf{nwd}(\mathcal{P}) := \max_{v \in V} \{|\lambda(v)|\},$$

that is, the size of the largest CQ that labels a node of T.

[10] We can assume that the auxiliary predicates introduced during the normalization of Σ do not occur in D or q.

Before we strengthen Theorem 4.9, let us define the polynomial that will allow us to bound the node-width of the linear proof trees that we need to consider. This polynomial relies on the notion of predicate level. Consider a set Σ of TGDs. For a predicate $P \in \mathsf{sch}(\Sigma)$, we write $\mathsf{rec}(P)$ for the set of predicates of $\mathsf{sch}(\Sigma)$ that are mutually recursive to P according to $\mathsf{pg}(\Sigma) = (V, E)$. Let $\ell_\Sigma \colon \mathsf{sch}(\Sigma) \to \mathbb{N}$ be the unique function that satisfies

$$\ell_\Sigma(P) \;=\; \begin{cases} 1 & \text{if } \{R \mid (R,P) \in E \text{ and } R \notin \mathsf{rec}(P)\} = \emptyset, \\ \\ \max\{\ell_\Sigma(R) \mid (R,P) \in E \text{ and } R \notin \mathsf{rec}(P)\} + 1 & \text{otherwise,} \end{cases}$$

with $\ell_\Sigma(P)$ being the *level of P w.r.t. Σ*, for each $P \in \mathsf{sch}(\Sigma)$. It is easy to verify that the function ℓ_Σ exists and is unique. Roughly speaking, for each weakly connected component C of $\mathsf{pg}(\Sigma)$, assuming that $C_1, \ldots, C_k$ is the (unique) topological sort of the strongly connected components of C, ℓ_Σ assigns the integer $i \in [k]$ to each node of C_i. We define the polynomial

$$f_{\mathsf{WARD} \cap \mathsf{PWL}}(q, \Sigma) \;:=\; (|q| + 1) \cdot \max_{P \in \mathsf{sch}(\Sigma)} \{\ell_\Sigma(P)\} \cdot \max_{\sigma \in \Sigma} \{|\mathsf{body}(\sigma)|\}\,.$$

To simplify our technical proofs even further, apart from assuming single-head TGDs (which can be done due to Lemma 4.10), we also assume, w.l.o.g., that the level of a predicate in the body of a TGD σ is ℓ or $\ell - 1$, where ℓ is the level of the predicate of the single-head of σ, i.e., we assume sets of TGDs in *level-wise normal form*. This relies on the following transformation. Consider a set of single-head TGDs $\Sigma \in \mathsf{PWL}$, and a TGD $\sigma \in \Sigma$ of the form

$$R_1(\bar{x}_1), \ldots, R_n(\bar{x}_n) \;\to\; \exists \bar{z}\, P(\bar{x}, \bar{z}),$$

where $\bar{x} \subseteq \bar{x}_1 \cup \cdots \cup \bar{x}_n$. If, for every $i \in [n]$, $0 \leq \ell_\Sigma(P) - \ell_\Sigma(R_i) \leq 1$, then we define $\mathsf{L}_\Sigma(\sigma)$ as σ, i.e., σ is already in level-wise normal form (w.r.t. Σ). Assume now that $1 \leq i_1 < \cdots < i_m \leq n$ are such that $k_j = \ell_\Sigma(P) - \ell_\Sigma(R_{i_j}) > 1$, for $j \in [m]$, namely the level-wise normal form is violated due to the body-predicates $R_{i_1}, \ldots, R_{i_m}$. We define $\mathsf{L}_\Sigma(\sigma)$ as the set of the following TGDs: for each $j \in [m]$,

$$\begin{aligned} R_{i_j}(\bar{x}_{i_j}) &\;\to\; R^{\sigma,1}_{i_j}(\bar{x}_{i_j}) \\ R^{\sigma,1}_{i_j}(\bar{x}_{i_j}) &\;\to\; R^{\sigma,2}_{i_j}(\bar{x}_{i_j}) \\ &\;\vdots \\ R^{\sigma,k_j-1}_{i_j}(\bar{x}_{i_j}) &\;\to\; R^{\sigma,k_j}_{i_j}(\bar{x}_{i_j}), \end{aligned}$$

and the TGD

$$\delta(R_1)(\bar{x}_1) \ldots, \delta(R_n)(\bar{x}_n) \;\to\; \exists \bar{z}\, P(\bar{x}, \bar{z}),$$

where, for each $i \in [n]$,

$$\delta(R_i) \;=\; \begin{cases} R^{\sigma,k_j}_i & \text{if } i = i_j \text{ for some } j \in [m], \\ \\ R_i & \text{otherwise.} \end{cases}$$

We finally define $\mathsf{N}_{\mathsf{lw}}(\Sigma)$, i.e., the normalization of Σ into level-wise normal form, as the set of TGDs $\bigcup_{\sigma \in \Sigma} \mathsf{L}_\Sigma(\sigma)$. It is not difficult ot verify that $\mathsf{N}_{\mathsf{lw}}(\Sigma)$ is indeed in level-wise normal form. The next easy lemma collects some useful facts about the normalization procedure $\mathsf{N}_{\mathsf{lw}}(\cdot)$:

Lemma 4.11. *Consider a set of single-head TGDs $\Sigma \in \mathsf{WARD} \cap \mathsf{PWL}$. The following hold:*

(1) $\mathsf{N}_{\mathsf{lw}}(\Sigma)$ can be computed in polynomial time in the size of Σ.
(2) $\mathsf{N}_{\mathsf{lw}}(\Sigma)$ is a set of single-head TGDs in $\mathsf{WARD} \cap \mathsf{PWL}$.

(3) Given a database D, and a CQ $q(\bar{x})$, $\mathsf{cert}(q, D, \Sigma) = \mathsf{cert}(q, D, \mathsf{N}_{\mathsf{lw}}(\Sigma))$.[11]

We are now ready to strengthen Theorem 4.9; a proof-sketch is given below, while the full proof can be found in the appendix.

Theorem 4.12. *Consider a database D, a set $\Sigma \in \mathsf{WARD} \cap \mathsf{PWL}$ of TGDs, a CQ $q(\bar{x})$, and a tuple $\bar{c} \in \mathsf{dom}(D)^{|\bar{x}|}$. With $\Sigma' = \mathsf{N}_{\mathsf{lw}}(\mathsf{N}_{\mathsf{sh}}(\Sigma))$, the following are equivalent:*

(1) $\bar{c} \in \mathsf{cert}(q, D, \Sigma)$.

(2) There is a linear proof tree $\mathcal{P}$ of q w.r.t. Σ' with $\mathsf{nwd}(\mathcal{P}) \leq f_{\mathsf{WARD}\cap\mathsf{PWL}}(q, \Sigma')$ such that $\bar{c} \in \mathcal{P}(D)$.

Warded sets of TGDs. Now, in the case of arbitrary (not necessarily piece-wise linear) warded sets of TGDs, we cannot focus only on linear proof trees. Nevertheless, we can still bound the node-width of the proof trees that we need to consider by the following polynomial, which, unsurprisingly, does not rely anymore on the notion of predicate level:

$$f_{\mathsf{WARD}}(q, \Sigma) \;:=\; 2 \cdot \max\left\{|q|, \max_{\sigma \in \Sigma}\{|\mathsf{body}(\sigma)|\}\right\}.$$

Theorem 4.9 can be strengthened as follows; a proof-sketch is given below, while the full proof can be found in the appendix..

Theorem 4.13. *Consider a database D, a set $\Sigma \in \mathsf{WARD}$ of TGDs, a CQ $q(\bar{x})$, and a tuple $\bar{c} \in \mathsf{dom}(D)^{|\bar{x}|}$. With $\Sigma' = \mathsf{N}_{\mathsf{sh}}(\Sigma)$, the following are equivalent:*

(1) $\bar{c} \in \mathsf{cert}(q, D, \Sigma)$.

(2) There exists a proof tree $\mathcal{P}$ of q w.r.t. Σ' with $\mathsf{nwd}(\mathcal{P}) \leq f_{\mathsf{WARD}}(q, \Sigma')$ such that $\bar{c} \in \mathcal{P}(D)$.

A Proof Sketch. Let us now provide some details on how Theorems 4.12 and 4.13 are shown, while the full proofs are deferred to the appendix. For both theorems, (2) implies (1) readily follows from Theorem 4.9. The non-trivial task is to show the other direction. The main ingredients of the proof can be intuitively described as follows:

- We first introduce the auxiliary notion of chase tree, which can be understood as a concrete instantiation of a proof tree. It serves as an intermediate structure between proof trees and chase sequences, which allows us to use the chase as our underlying technical tool. Note that the notions of linearity and node-width can be naturally defined for chase trees.
- We then show that, if the given tuple of constants $\bar{c}$ is a certain answer to the given CQ q w.r.t. the given database D and (piece-wise linear) warded set Σ of TGDs,[12] then there is a (linear) chase tree for the image of q in $\mathsf{chase}(D, \Sigma)$, which exists due to Proposition 2.1, such that its node-width respects the bounds given above (see Lemmas 4.15 and 4.16).
- We finally show that the existence of a (linear) chase tree for the homomorphic image of q in $\mathsf{chase}(D, \Sigma)$ with node-width at most m implies the existence of a (linear) proof tree $\mathcal{P}$ of q w.r.t. Σ with node-width at most m such that $\bar{c} \in \mathcal{P}(D)$ (see Lemma 4.17).

Let us make the above description a bit more precise, while the missing details can be found in the appendix. In order to introduce the notion of chase tree, we first need to recall the notion of chase graph, then introduce the notion of unraveling of the chase graph, and finally introduce the notions of unfolding and decomposition for sets of atoms in the unraveling of the chase graph.

Fix a chase sequence $\delta = (I_i)_{i \geq 0}$ for a database D under a set Σ of TGDs with $I_i \langle \sigma_i, h_i \rangle I_{i+1}$, i.e., I_{i+1} is obtained by applying the trigger (σ_i, h_i) to I_i. The *chase graph* for D and Σ (w.r.t. δ) is a directed edge-labeled graph $\mathcal{G}^{D,\Sigma} = (V, E, \lambda)$, with λ being the labeling function, where $V = \mathsf{chase}(D, \Sigma)$,

[11] We can assume that the auxiliary predicates introduced during the level-wise normalization of Σ do not occur in D or q.

[12] For brevity, we assume here that the given (piece-wise linear) warded set Σ of TGDs is already in the proper normal form.

and an edge (α, β) labeled with (σ_k, h_k) belongs to E iff $\alpha \in h_k(\text{body}(\sigma_k))$ and $\beta \in I_{k+1} \setminus I_k$, for some $k \geq 0$. In simple words, α has an edge to β if β is derived using α, and, in addition, β is new in the sense that it has not been derived before. Notice that $\mathcal{G}^{D,\Sigma}$ has no directed cycles. It is clear that $\mathcal{G}^{D,\Sigma}$ depends on δ, but we can assume a fixed sequence δ since, as discussed in Section 2, every chase sequence leads to the same result (up to isomorphism).

We now discuss the notion of unraveling of the chase graph; we keep this discussion informal, while the details are deferred to the appendix. For a set $\Theta \subseteq \text{chase}(D, \Sigma)$, the *unraveling of* $\mathcal{G}^{D,\Sigma}$ *around* Θ is a directed node- and edge-labeled forest $\mathcal{G}^{D,\Sigma}_\Theta$ that has a tree for each $\alpha \in \Theta$ whose branches are essentially backward-paths in $\mathcal{G}^{D,\Sigma}$ from α to a database atom. Intuitively, $\mathcal{G}^{D,\Sigma}_\Theta$ is a forest-like reorganization of the atoms of $\text{chase}(D, \Sigma)$ that are needed to derive Θ. Due to its forest-like shape, it may contain multiple copies of atoms of $\text{chase}(D, \Sigma)$. The edges between nodes are labeled by pairs (σ, h) just like in $\mathcal{G}^{D,\Sigma}$, while the nodes are labeled by atoms and, importantly, the atoms along the paths in $\mathcal{G}^{D,\Sigma}$ may be duplicated and labeled nulls are given new names. We write $U(\mathcal{G}^{D,\Sigma}, \Theta)$ for the set of all atoms that appear as node labels in $\mathcal{G}^{D,\Sigma}_\Theta$, and $\text{succ}_{\sigma,h}(v)$ for the set of children of a node v of $\mathcal{G}^{D,\Sigma}_\Theta$ whose incoming edge is labeled with (σ, h). It is important to say that there exists a homomorphism h_Θ that maps Θ to $U(\mathcal{G}^{D,\Sigma}, \Theta)$.

Let us now introduce the notions of unfolding and decomposition. For sets $\Gamma, \Gamma' \subseteq U(\mathcal{G}^{D,\Sigma}, \Theta)$, we say that Γ' *is an unfolding of* Γ if there are $\alpha \in \Gamma$ and $\beta_1, \ldots, \beta_k \in U(\mathcal{G}^{D,\Sigma}, \Theta)$ such that

(1) $\text{succ}_{\sigma,h}(v) = \{\beta_1, \ldots, \beta_k\}$, for some $\sigma \in \Sigma$ and h, and some node v of $\mathcal{G}^{D,\Sigma}_\Theta$ labeled with α,
(2) for every null that occurs in α, either it does not appear in $\Gamma \setminus \{\alpha\}$, or it appears in $\{\beta_1, \ldots, \beta_k\}$,
(3) $\Gamma' = (\Gamma \setminus \{\alpha\}) \cup \{\beta_1, \ldots, \beta_k\}$.

Let $\Gamma \subseteq U(\mathcal{G}^{D,\Sigma}, \Theta)$ be a non-empty set. A *decomposition* of Γ is a set $\{\Gamma_1, \ldots, \Gamma_n\}$, for $n \geq 1$, of non-empty subsets of Γ such that (i) $\Gamma = \bigcup_{i \in [n]} \Gamma_i$, and (ii) $i \neq j$ implies that Γ_i and Γ_j do not share a labeled null. We can now define the key notion of chase tree:

*Definition 4.14 (**Chase Tree**).* Consider a database D, a set Σ of TGDs, a set $\Theta \subseteq \text{chase}(D, \Sigma)$, and a finite set $\Gamma \subseteq U(\mathcal{G}^{D,\Sigma}, \Theta)$. A *chase tree* for Γ (w.r.t. $\mathcal{G}^{D,\Sigma}_\Theta$) is a pair $\mathcal{C} = (T, \lambda)$, where $T = (V, E)$ is a finite rooted tree, and λ a labeling function that assigns a subset of $U(\mathcal{G}^{D,\Sigma}, \Theta)$ to each node of T, such that, for each $v \in V$, the following hold:

(1) If v is the root node of T, then $\lambda(v) = \Gamma$.
(2) If v has only one child u, $\lambda(u)$ is an unfolding of $\lambda(v)$.
(3) If v has children $u_1, \ldots, u_k$ for $k > 1$, then $\{\lambda(u_1), \ldots, \lambda(u_k)\}$ is a decomposition of $\lambda(v)$.
(4) If v is a leaf node, then $\lambda(v) \subseteq D$.

The *node-width* of $\mathcal{C}$ is defined as $\text{nwd}(\mathcal{C}) := \max_{v \in V}\{|\lambda(v)|\}$. Moreover, we say that $\mathcal{C}$ is *linear* if, for each $v \in V$, there exists at most one $u \in V$ such that $(v, u) \in E$ and u is not a leaf. ∎

We can now state our auxiliary technical lemmas. The first one states that in the case of piece-wise linear warded sets of TGDs, we can always find a linear chase tree for a finite set of atoms in the unravelling such that its node-width respects the desired bound.

Lemma 4.15. *Consider a database D, and a set $\Sigma \in \text{WARD} \cap \text{PWL}$ of single-head TGDs in level-wise normal form. Let $\Theta \subseteq \text{chase}(D, \Sigma)$ and $\Gamma \subseteq U(\mathcal{G}^{D,\Sigma}, \Theta)$. There exists a linear chase tree $\mathcal{C}$ for Γ such that $\text{nwd}(\mathcal{C}) \leq f_{\text{WARD}\cap\text{PWL}}(\Gamma, \Sigma)$.*

The next lemma states an analogous result for arbitrary warded sets of TGDs.

Lemma 4.16. *Consider a database D, and a set $\Sigma \in \text{WARD}$ of single-head TGDs. Let $\Theta \subseteq \text{chase}(D, \Sigma)$ and $\Gamma \subseteq U(\mathcal{G}^{D,\Sigma}, \Theta)$. There exists a chase tree $\mathcal{C}$ for Γ such that $\text{nwd}(\mathcal{C}) \leq f_{\text{WARD}}(\Gamma, \Sigma)$.*

The next technical lemma exposes the connection between chase trees and proof trees:

Algorithm 1: Non-deterministic algorithm for CQAns(WARD ∩ PWL)

Input: A database D, a set of TGDs $\Sigma \in \mathsf{WARD} \cap \mathsf{PWL}$, a CQ $q(\bar{x})$, and a tuple $\bar{c} \in \mathsf{dom}(D)^{|\bar{x}|}$
Output: Accept if $\bar{c} \in \mathsf{cert}(q, D, \Sigma)$; otherwise, Reject

$\Sigma := \mathsf{N}_{\mathsf{lw}}(\mathsf{N}_{\mathsf{sh}}(\Sigma))$
$p := Q \leftarrow \alpha_1, \ldots, \alpha_n$ with $\mathsf{atoms}(q(\bar{c})) = \{\alpha_1, \ldots, \alpha_n\}$
repeat
  if $\mathsf{atoms}(p) \subseteq D$ **then**
    Accept
  guess $op \in \{\mathsf{r}, \mathsf{d}, \mathsf{s}\}$
  if $op = \mathsf{r}$ **then**
    guess a TGD $\sigma \in \Sigma$
    if $\mathsf{mgcu}(p, \sigma) = \emptyset$ **then**
      Reject
    else
      guess $U \in \mathsf{mgcu}(p, \sigma)$
      if $|p[\sigma, U]| > f_{\mathsf{WARD} \cap \mathsf{PWL}}(q, \Sigma)$ **then**
        Reject
      else
        $p' := p[\sigma, U]$
  if $op = \mathsf{d}$ **then**
    $p' := p[-D]$
  if $op = \mathsf{s}$ **then**
    guess $V \subseteq \mathsf{var}(p)$ and $\gamma : V \to \mathsf{dom}(D)$
    $p' := \gamma(p)$
  $p := p'$
until False;

Lemma 4.17. *Consider a database D and a set Σ of TGDs. Let $\Theta \subseteq \mathsf{chase}(D, \Sigma)$, $q(\bar{x})$ be a CQ, and $\bar{c}$ be a tuple of constants such that $h(\mathsf{atoms}(q)) \subseteq U(\mathcal{G}^{D,\Sigma}, \Theta)$ and $h(\bar{x}) = \bar{c}$, for some homomorphism h. If there exists a (linear) chase tree C for $h(\mathsf{atoms}(q))$ with $\mathsf{nwd}(C) \leq m$, then there exists a (linear) proof tree $\mathcal{P}$ for q w.r.t. Σ such that $\mathsf{nwd}(\mathcal{P}) \leq m$ and $\bar{c} \in \mathcal{P}(D)$.*

We can now show the direction (1) implies (2) of Theorem 4.12 – the same for Theorem 4.13 can be shown analogously by using Lemma 4.16 instead of Lemma 4.15. Consider a database D, a set of TGDs $\Sigma \in \mathsf{WARD} \cap \mathsf{PWL}$, a CQ $q(\bar{x})$, and a tuple $\bar{c} \in \mathsf{dom}(D)^{|\bar{x}|}$. Assume that $\bar{c} \in \mathsf{cert}(q, D, \Sigma)$, which in turn implies that $\bar{c} \in \mathsf{cert}(q, D, \Sigma')$ with $\Sigma' = \mathsf{N}_{\mathsf{lw}}(\mathsf{N}_{\mathsf{sh}}(\Sigma))$. We need to show that there exists a linear proof tree $\mathcal{P}$ of q w.r.t. Σ' with $\mathsf{nwd}(\mathcal{P}) \leq f_{\mathsf{WARD} \cap \mathsf{PWL}}(q, \Sigma')$ such that $\bar{c} \in \mathcal{P}(D)$. By hypothesis and Proposition 2.1, there is a homomorphism h such that $h(\mathsf{atoms}(q)) \subseteq \mathsf{chase}(D, \Sigma')$ and $h(\bar{x}) = \bar{c}$. Let Θ_q be the set $h(\mathsf{atoms}(q))$. Recall that there is a homomorphism h_{Θ_q} that maps Θ_q to $U(\mathcal{G}^{D,\Sigma'}, \Theta_q)$. Thus, $h' = h_{\Theta_q} \circ h$ is such that $h'(\mathsf{atoms}(q)) \subseteq U(\mathcal{G}^{D,\Sigma'}, \Theta_q)$ and $h'(\bar{x}) = \bar{c}$. By Lemma 4.15, there exists a chase tree C for $h'(\mathsf{atoms}(q))$ with $\mathsf{nwd}(C) \leq f_{\mathsf{WARD} \cap \mathsf{PWL}}(h'(\mathsf{atoms}(q)), \Sigma)$. By Lemma 4.17, we get that there exists a linear proof tree $\mathcal{P}$ of q w.r.t Σ with

$$\mathsf{nwd}(\mathcal{P}) \;\leq\; f_{\mathsf{WARD} \cap \mathsf{PWL}}(h'(\mathsf{atoms}(q)), \Sigma') \;\leq\; f_{\mathsf{WARD} \cap \mathsf{PWL}}(q, \Sigma')$$

such that $\bar{c} \in \mathcal{P}(D)$, and the claim follows.

4.4 Complexity Analysis

We now have all the tools for showing that CQ answering under piece-wise linear warded sets of TGDs is in PSpace in combined complexity, and in NLogSpace in data complexity, and also for re-establishing the complexity of warded sets of TGDs (see Proposition 3.2) in a more transparent and elegant way than the approach followed in [3, 18].

Piece-wise Linear Warded Sets of TGDs. Given a database D, a set $\Sigma \in \mathsf{WARD} \cap \mathsf{PWL}$ of TGDs, a CQ $q(\bar{x})$, and a tuple $\bar{c} \in \mathsf{dom}(D)^{|\bar{x}|}$, by Theorem 4.12, our problem boils down to checking whether there exists a linear proof tree $\mathcal{P}$ of q w.r.t. Σ', where $\Sigma' = \mathsf{N}_{\mathsf{lw}}(\mathsf{N}_{\mathsf{sh}}(\Sigma))$, with $\mathsf{nwd}(\mathcal{P}) \leq f_{\mathsf{WARD}\cap\mathsf{PWL}}(q, \Sigma')$ such that $\bar{c} \in \mathcal{P}(D)$. This can be easily checked via a space-bounded algorithm that is trying to build such a proof tree in a level-by-level fashion. Essentially, the algorithm builds the i-th level from the $(i-1)$-th level of the proof tree by non-deterministically applying the operations introduced above, i.e., resolution, decomposition and specialization.

The algorithm is depicted in Algorithm 1. Here is a semi-formal description of it. The first step is to normalize Σ into a set of single-head TGDs in level-wise normal form, and also store in p the Boolean CQ obtained after instantiating the output variables of q with $\bar{c}$. The rest of the algorithm is an iterative procedure that non-deterministically constructs p' (the i-th level) from p (the $(i-1)$-th level) until it reaches a level that is a subset of the database D. Notice that p and p' always hold one CQ since at each level of a linear proof tree only one node has a child, while all the other nodes are leaves, which essentially means that their atoms appear in the database D. At each iteration, the algorithm constructs p' from p by applying one of the main operations underlying proof trees, namely resolution (r), decomposition (d), or specialization (s):

Resolution. It guesses a TGD $\sigma \in \Sigma$. If the set $\mathsf{mgcu}(p, \sigma)$, i.e., the set of all MGCUs of p with σ, is empty, then rejects; otherwise, it guesses $U \in \mathsf{mgcu}(p, \sigma)$. If the size of the σ-resolvent of p obtained via U, denoted $p[\sigma, U]$, exceeds the bound provided by Theorem 4.12, then it rejects; otherwise, it assigns $p[\sigma, U]$ to p'. Recall that during a resolution step we need to rename variables in order to avoid undesirable clutter. However, we cannot blindly use new variables at each step since this will explode the space used by the algorithm. Instead, we should reuse variables that have been lost due to their unification with an existentially quantified variable. We only need polynomially many variables, while this polynomial depends only on q and Σ.

Decomposition. It deletes from p the atoms that occur in D, and it assigns the obtained CQ $p[-D]$ to p'. Notice that $p[-D]$ may be empty in case $\mathsf{atoms}(p) \subseteq D$. Essentially, the algorithm decomposes p in such a way that the subquery of p consisting of $\mathsf{atoms}(p) \cap D$ forms a child of p that is a leaf, while the subquery consisting of $\mathsf{atoms}(p) \setminus D$ is the non-leaf child.

Specialization. It assigns to p' a specialized version of p, where some variables are instantiated by constants of $\mathsf{dom}(D)$. The convention that output variables correspond to constants is implemented by directly instantiating them with actual constants from $\mathsf{dom}(D)$.

After constructing p', the algorithm assigns it to p, and this ends one iteration. The first step of the next iteration is to check whether $\mathsf{atoms}(p) \subseteq D$, in which case a linear proof tree $\mathcal{P}$ such that $\bar{c} \in \mathcal{P}(D)$ has been found, and the algorithm accepts; otherwise, it proceeds further.

It is easy to see that the algorithm uses polynomial space in general. Moreover, in case the set of TGDs and the CQ are fixed, the algorithm uses logarithmic space, which is the space needed for representing constantly many elements of $\mathsf{dom}(D)$; each element of $\mathsf{dom}(D)$ can be represented using logaritmically many bits. The desired upper bounds claimed in Theorem 4.2 follow.

Warded Sets of TGDs. The non-deterministic algorithm discussed above cannot be directly used for warded sets of TGDs since it is not enough to search for a linear proof tree as in the case of

Algorithm 2: Alternating algorithm for CQAns(WARD)

Input: A database D, a set of TGDs $\Sigma \in$ WARD, a CQ $q(\bar{x})$, and a tuple $\bar{c} \in \mathsf{dom}(D)^{|\bar{x}|}$
Output: Accept if $\bar{c} \in \mathsf{cert}(q, D, \Sigma)$; otherwise, Reject

$\Sigma := \mathsf{N}_{\mathsf{sh}}(\Sigma)$
$p := Q \leftarrow \alpha_1, \ldots, \alpha_n$ with $\mathsf{atoms}(q(\bar{c})) = \{\alpha_1, \ldots, \alpha_n\}$
repeat
  if $\mathsf{atoms}(p) \subseteq D$ **then**
    Accept
  guess $op \in \{\mathsf{r}, \mathsf{d}, \mathsf{s}\}$
  if $op = \mathsf{r}$ **then**
    guess a TGD $\sigma \in \Sigma$
    if $\mathsf{mgcu}(p, \sigma) = \emptyset$ **then**
      Reject
    else
      guess $U \in \mathsf{mgcu}(p, \sigma)$
      if $|p[\sigma, U]| > f_{\mathsf{WARD}}$ **then**
        Reject
      else
        $P := \{p[\sigma, U]\}$
  if $op = \mathsf{d}$ **then**
    guess a decomposition P of p
  if $op = \mathsf{s}$ **then**
    guess $V \subseteq \mathsf{var}(p)$ and $\gamma : V \rightarrow \mathsf{dom}(D)$
    $P := \{\gamma(p)\}$
  universally select every CQ $p \in P$
until False;

piece-wise linear warded sets of TGDs. However, by Theorem 4.13, we can search for a proof tree that has bounded node-width. This allows us to devise a space-bounded algorithm, which is similar in spirit to Algorithm 1, with the crucial difference that it constructs in a level-by-level fashion the branches of the proof tree in parallel universal computations using alternation. This algorithm is shown in Algorithm 2. The key difference compared to Algorithm 1 is that each iteration constructs a set of CQs P, instead of one CQ p', and at the end each CQ of P is universally selected. Since this alternating algorithm uses polynomial space in general, and logarithmic space when the set of TGDs and the CQ are fixed, we immediately get an ExpTime upper bound in combined, and a PTime upper bound in data complexity. This confirms Proposition 3.2 established in [3, 18]. However, our new algorithm is significantly simpler than the one employed in [3, 18], while Theorem 4.13 reveals the main property of warded sets of TGDs that leads to the desirable complexity upper bounds.

5 A JUSTIFIED COMBINATION

It is interesting to observe that the class of piece-wise linear warded sets of TGDs generalizes the class of *intensionally linear* sets of TGDs, denoted IL, where each TGD has at most one body atom whose predicate is intensional. Therefore, Theorem 4.2 immediately implies that CQAns(IL) is PSpace-complete in combined complexity, and NLogSpace-complete in data complexity. Notice that IL generalizes linear Datalog, which is also PSpace-complete in combined complexity, and NLogSpace-complete in data complexity. Therefore, we can extend linear Datalog by allowing existentially quantified variables in rule heads, which essentially leads to IL, without affecting the complexity of query answering. At this point, one may be tempted to think that the same

holds for piece-wise linear Datalog, i.e., we can extend it with existentially quantified variables in rule heads, which leads to PWL, without affecting the complexity of query answering, that is, PSPACE-complete in combined, and NLOGSPACE-complete in data complexity. However, if this is the case, then wardedness becomes redundant since the formalism that we are looking for is the class of piece-wise linear sets of TGDs, without the wardedness condition. It turns out that this is not the case. To our surprise, CQAns(PWL) is undecidable even in data complexity:

THEOREM 5.1. *There exists a set* $\Sigma \in \mathsf{PWL}$ *and a CQ* q *such that* $\mathsf{CQAns}(\Sigma, q)$ *is undecidable.*

The rest of this section is devoted to showing the above result. To this end, we exploit a well-known undecidable tiling problem [13]. A *tiling system* is a tuple $\mathbb{T} = (T, L, R, H, V, a, b)$, where T is a finite set of tiles, $L, R \subseteq T$ are special sets of left and right border tiles, respectively, with $L \cap R = \emptyset$, $H, V \subseteq T^2$ are the horizontal and vertical constraints, and a, b are distinguished tiles of T called the start and the finish tile, respectively. A *tiling* for $\mathbb{T}$ is a function $f : [n] \times [m] \to T$, for some $n, m > 0$, such that $f(1,1) = a$, $f(1,m) = b$, $f(1,i) \in L$ and $f(n,i) \in R$, for every $i \in [m]$, and f respects the horizontal and vertical constraints. In other words, (i) the first and the last rows of a tiling for $\mathbb{T}$ start with a and b, respectively, (ii) the leftmost and rightmost columns contain only tiles from L and R, respectively, (iii) for every two consecutive tiles t, t' in a row, i.e., t occurs on the left of t', it holds that $(t, t') \in H$, and (iv) for every two consecutive tiles t, t' in a column, i.e., t occurs on top of t', it holds that $(t, t') \in V$. We reduce from the UnboundedTiling problem, that is, given a tiling system $\mathbb{T}$, decide whether there is a tiling for $\mathbb{T}$. More precisely, the goal is to show that there exists a set of TGDs $\Sigma \in \mathsf{PWL}$, and a Boolean CQ q, such that the following holds: given a tiling system $\mathbb{T} = (T, L, R, H, V, a, b)$, we can construct in polynomial time a database $D_{\mathbb{T}}$ such that $\mathbb{T}$ has a tiling iff $() \in \mathsf{cert}(q, D_{\mathbb{T}}, \Sigma)$, where $()$ denotes the empty tuple.

The Database $D_{\mathbb{T}}$. It simply stores the tiling system $\mathbb{T}$:

$$\begin{aligned} & \{\mathrm{Tile}(t) \mid t \in T\} \cup \{\mathrm{Left}(t) \mid t \in L\} \cup \{\mathrm{Right}(t) \mid t \in R\} \\ \cup \quad & \{H(t,t') \mid (t,t') \in H\} \cup \{V(t,t') \mid (t,t') \in V\} \\ \cup \quad & \{\mathrm{Start}(a), \mathrm{Finish}(b)\}. \end{aligned}$$

The Set of TGDs Σ. It is responsible for generating all the candidate tilings for $\mathbb{T}$, i.e., tilings without the condition $f(1,m) = b$, of arbitrary width and depth. Whether there exists a candidate tiling for $\mathbb{T}$ that satisfies the condition $f(1,m) = b$ will be checked by the CQ q. The set Σ essentially implements the following idea: construct rows of size ℓ from rows of size $\ell - 1$, for $\ell > 1$, that respect the horizontal constraints, and then construct all the candidate tilings by combining compatible rows, i.e., rows that respect the vertical constraints. A row r is encoded as an atom $\mathrm{Row}(p, c, s, e)$, where p is the id of the row from which r has been obtained, i.e., the previous one, c is the id of r, i.e., the current one, s is the starting tile of r, and e is the ending tile of r. We write $\mathrm{Row}(c, c, s, s)$ for rows consisting of a single tile, which do not have a previous row (hence the id of the previous row coincides with the id of the current row), and the starting tile is the same as the ending tile. The following two TGDs construct all the rows that respect the horizontal constraints:

$$\begin{aligned} \mathrm{Tile}(x) &\to \exists z\, \mathrm{Row}(z, z, x, x), \\ \mathrm{Row}(\_, x, y, z), H(z, w) &\to \exists u\, \mathrm{Row}(x, u, y, w). \end{aligned}$$

Similarly to Prolog, we write "_" for a "don't-care" variable that occurs only once in the TGD. The next set of TGDs constructs all the pairs of compatible rows, i.e., pairs of rows (r_1, r_2) such that we can place r_2 below r_1 without violating the vertical constraints. This is done inductively as follows:

$$\begin{aligned} \mathrm{Row}(x, x, y, y), \mathrm{Row}(x', x', y', y'), V(y, y') &\to \mathrm{Comp}(x, x'), \\ \mathrm{Row}(x, y, \_, z), \mathrm{Row}(x', y', \_, z'), \mathrm{Comp}(x, x'), V(z, z') &\to \mathrm{Comp}(y, y'). \end{aligned}$$

We finally compute all the candidate tilings, together with their bottom-left tile, using the TGDs:

$$\mathrm{Row}(\_, x, y, z), \mathrm{Start}(y), \mathrm{Right}(z) \rightarrow \mathrm{CTiling}(x, y),$$
$$\mathrm{CTiling}(x, \_), \mathrm{Row}(\_, y, z, w), \mathrm{Comp}(x, y), \mathrm{Left}(z), \mathrm{Right}(w) \rightarrow \mathrm{CTiling}(y, z).$$

This concludes the definition of Σ.

The Boolean CQ q. Recall that q is responsible for checking whether there exists a candidate tiling such that its bottom-left tile is b. This can be easily done via the query

$$Q \leftarrow \mathrm{CTiling}(x, y), \mathrm{Finish}(y).$$

It is easy to verify the following lemma, which immediately implies Theorem 5.1.

Lemma 5.2. *It hold that:*

(1) $D_{\mathbb{T}}$ can be constructed in polynomial time in the size of $\mathbb{T}$.
(2) $\Sigma \in \mathsf{PWL}$.
(3) There exists a tiling for $\mathbb{T}$ iff $() \in \mathsf{cert}(q, D_{\mathbb{T}}, \Sigma)$.

6 EXPRESSIVE POWER

A class of TGDs naturally gives rise to a declarative database query language. More precisely, we consider queries of the form (Σ, q), where Σ is a set of TGDs, and q a CQ over $\mathsf{sch}(\Sigma)$.[13] The *extensional (database) schema* of Σ, denoted $\mathsf{edb}(\Sigma)$, is the set of extensional predicates of $\mathsf{sch}(\Sigma)$, i.e., the predicates that do not occur in the head of a TGD of Σ. The *intensional schema* of Σ, denoted $\mathsf{idb}(\Sigma)$, is the set of intensional predicates of Σ, that is, $\mathsf{sch}(\Sigma) \setminus \mathsf{edb}(\Sigma)$. Given a query $Q = (\Sigma, q)$ and a database D over $\mathsf{edb}(\Sigma)$, the *evaluation* of Q over D, denoted $Q(D)$, is defined as $\mathsf{cert}(q, D, \Sigma)$. We write $(\mathsf{C}, \mathsf{CQ})$ for the query language consisting of all the queries (Σ, q), where $\Sigma \in \mathsf{C}$, and q is a CQ. The evaluation problem for $(\mathsf{C}, \mathsf{CQ})$, dubbed $\mathsf{Eval}(\mathsf{C}, \mathsf{CQ})$, is defined as expected:

PROBLEM :	$\mathsf{Eval}(\mathsf{C}, \mathsf{CQ})$
INPUT :	A query $Q = (\Sigma, q(\bar{x}))$ from $(\mathsf{C}, \mathsf{CQ})$, a database D over $\mathsf{edb}(\Sigma)$, and a tuple $\bar{c} \in \mathsf{dom}(D)^{\lvert\bar{x}\rvert}$.
QUESTION :	Is it the case that $\bar{c} \in Q(D)$?

This general formulation refers to the combined complexity of the problem. We can also refer to its data complexity, which measures the complexity of the problem assuming that the query Q is fixed. It should be clear that the complexity of $\mathsf{Eval}(\mathsf{C}, \mathsf{CQ})$ when $\mathsf{C} = \mathsf{WARD} \cap \mathsf{PWL}$ and $\mathsf{C} = \mathsf{WARD}$ is immediately inherited from Theorem 4.2 and Proposition 3.2, respectively.

Theorem 6.1. *The following statements hold:*

(1) $\mathsf{Eval}(\mathsf{WARD} \cap \mathsf{PWL}, \mathsf{CQ})$ is PSpace-complete in combined complexity, and NLogSpace-complete in data complexity.
(2) $\mathsf{Eval}(\mathsf{WARD}, \mathsf{CQ})$ is ExpTime-complete in combined complexity, and PTime-complete in data complexity.

The main goal of this section is to understand the relative expressive power of $(\mathsf{WARD} \cap \mathsf{PWL}, \mathsf{CQ})$ and $(\mathsf{WARD}, \mathsf{CQ})$. To this end, we are going to adopt two different notions of expressive power: the classical one, which we call combined expressive power since it considers the set of TGDs and the CQ as one composite query, and the program expressive power, which aims at the decoupling of the set of TGDs from the actual CQ. We start with the combined expressive power.

[13] Such queries are also known in the literature as ontology-mediated queries [12].

6.1 Combined Expressive Power

Consider a query $Q = (\Sigma, q)$, where Σ is a set of TGDs and $q(\bar{x})$ a CQ over $\mathsf{sch}(\Sigma)$. The *expressive power* of Q, denoted $\mathsf{ep}(Q)$, is the set of pairs $(D, \bar{c})$, where D is a database over $\mathsf{edb}(\Sigma)$, and $\bar{c}$ is a tuple from $\mathsf{dom}(D)^{|\bar{x}|}$ such that $\bar{c} \in Q(D)$. The *combined expressive power* of a query language $(\mathsf{C}, \mathsf{CQ})$, where C is a class of TGDs, is defined as the set of database-tuple pairs

$$\mathsf{cep}(\mathsf{C}, \mathsf{CQ}) \;=\; \{\mathsf{ep}(Q) \mid Q \in (\mathsf{C}, \mathsf{CQ})\}.$$

Given two query languages $\mathsf{Q}_1, \mathsf{Q}_2$, we say that Q_2 is *more expressive (w.r.t. the combined expressive power)* than Q_1, written $\mathsf{Q}_1 \leq_{\mathsf{cep}} \mathsf{Q}_2$, if $\mathsf{cep}(\mathsf{Q}_1) \subseteq \mathsf{cep}(\mathsf{Q}_2)$. We say that Q_1 and Q_2 are *equally expressive (w.r.t. the combined expressive power)*, written $\mathsf{Q}_1 =_{\mathsf{cep}} \mathsf{Q}_2$, if $\mathsf{Q}_1 \leq_{\mathsf{cep}} \mathsf{Q}_2$ and $\mathsf{Q}_2 \leq_{\mathsf{cep}} \mathsf{Q}_1$. It is easy to show that $\mathsf{Q}_1 =_{\mathsf{cep}} \mathsf{Q}_2$ is equivalent to saying that every query of Q_1 can be equivalently rewritten as a query of Q_2, and vice versa. We write $\mathsf{Q}_1 \preceq \mathsf{Q}_2$ if, for every $Q = (\Sigma, q) \in \mathsf{Q}_1$, there exists $Q' = (\Sigma', q') \in \mathsf{Q}_2$ such that, for every D over $\mathsf{edb}(\Sigma)$, $Q(D) = Q'(D)$.

Lemma 6.2. *Consider two query languages* Q_1 *and* Q_2*. It holds that* $\mathsf{Q}_1 \leq_{\mathsf{cep}} \mathsf{Q}_2$ *iff* $\mathsf{Q}_1 \preceq \mathsf{Q}_2$.

Proof. Consider a query $Q = (\Sigma, q) \in \mathsf{Q}_1$. By hypothesis, $\mathsf{cep}(\mathsf{Q}_1) \subseteq \mathsf{cep}(\mathsf{Q}_2)$, which in turn implies that $\mathsf{ep}(Q) \in \mathsf{cep}(\mathsf{Q}_2)$. Hence, there is a query $Q' = (\Sigma', q') \in \mathsf{Q}_2$ such that $\mathsf{ep}(Q) = \mathsf{ep}(Q')$. Thus, for every database D over $\mathsf{edb}(\Sigma)$, $Q(D) = Q'(D)$, which implies that $\mathsf{Q}_1 \preceq \mathsf{Q}_2$. Conversely, consider a set of database-tuple pairs $\mathsf{ep}(Q) \in \mathsf{cep}(\mathsf{Q}_1)$, where $Q \in \mathsf{Q}_1$. By hypothesis, $\mathsf{Q}_1 \preceq \mathsf{Q}_2$, which in turn implies that there is a query $Q' = (\Sigma', q') \in \mathsf{Q}_2$ such that, for every database D over $\mathsf{edb}(\Sigma)$, $Q(D) = Q'(D)$. Therefore, $\mathsf{ep}(Q) = \mathsf{ep}(Q')$, which implies that $\mathsf{ep}(Q) \in \mathsf{cep}(\mathsf{Q}_2)$. □

We are now ready to state the main result of this section, which reveals the expressiveness of $(\mathsf{WARD} \cap \mathsf{PWL}, \mathsf{CQ})$ and $(\mathsf{WARD}, \mathsf{CQ})$ relative to Datalog. Let us clarify that a Datalog query is actually a pair (Σ, q), where Σ is a Datalog program, or a set of single-head *full* TGDs, that is, single-head TGDs without existentially quantified variables – we write FULL_1 for this class – and q a CQ over $\mathsf{sch}(\Sigma)$. In other words, piece-wise linear Datalog, denoted PWL-DATALOG, is the language $(\mathsf{FULL}_1 \cap \mathsf{PWL}, \mathsf{CQ})$, while Datalog, denoted DATALOG, is the language $(\mathsf{FULL}_1, \mathsf{CQ})$, and thus we can refer to their combined expressive power. In what follows, PWL-DATALOG and $(\mathsf{FULL}_1 \cap \mathsf{PWL}, \mathsf{CQ})$ (resp., DATALOG and $(\mathsf{FULL}_1, \mathsf{CQ})$) are used interchangeably.

Theorem 6.3. *The following statements hold:*

(1) $\mathsf{PWL\text{-}DATALOG} =_{\mathsf{cep}} (\mathsf{WARD} \cap \mathsf{PWL}, \mathsf{CQ})$.
(2) $\mathsf{DATALOG} =_{\mathsf{cep}} (\mathsf{WARD}, \mathsf{CQ})$.

It is easy to verify that $\mathsf{FULL}_1 \cap \mathsf{PWL} \subseteq \mathsf{WARD} \cap \mathsf{PWL}$, which implies that $(\mathsf{FULL}_1 \cap \mathsf{PWL}, \mathsf{CQ}) \preceq (\mathsf{WARD} \cap \mathsf{PWL}, \mathsf{CQ})$. Analogously, $\mathsf{FULL}_1 \subseteq \mathsf{WARD}$, and thus $(\mathsf{FULL}_1, \mathsf{CQ}) \preceq (\mathsf{WARD}, \mathsf{CQ})$. Therefore, by Lemma 6.2, $\mathsf{PWL\text{-}DATALOG} \leq_{\mathsf{cep}} (\mathsf{WARD} \cap \mathsf{PWL}, \mathsf{CQ})$ and $\mathsf{DATALOG} \leq_{\mathsf{cep}} (\mathsf{WARD}, \mathsf{CQ})$. To conclude the proof of Theorem 6.3, Lemma 6.2 tells us that it suffices to show the next result.

Lemma 6.4. *The following statements hold:*

(1) $(\mathsf{WARD} \cap \mathsf{PWL}, \mathsf{CQ}) \preceq \mathsf{PWL\text{-}DATALOG}$.
(2) $(\mathsf{WARD}, \mathsf{CQ}) \preceq \mathsf{DATALOG}$.

The key underlying Lemma 6.4 is that we can convert a (linear) proof tree $\mathcal{P}$ of a CQ $q(\bar{x})$ w.r.t. a set Σ of TGDs into a (piece-wise linear) Datalog query $Q_{\mathcal{P}} = (\Sigma_{\mathcal{P}}, q_{\mathcal{P}}(\bar{x}))$ such that, for every database D over $\mathsf{edb}(\Sigma)$, $\mathcal{P}(D) = Q_{\mathcal{P}}(D)$. This, together with the fact that for (piece-wise linear) warded sets of TGDs it suffices to consider proof trees of bounded node-width, allow us to effectively rewrite every query $Q \in (\mathsf{WARD} \cap \mathsf{PWL}, \mathsf{CQ})$ (resp., $Q \in (\mathsf{WARD}, \mathsf{CQ})$) into an equivalent query that falls in PWL-DATALOG (resp., DATALOG). We first explain how a (linear) proof tree can be

converted into a (piece-wise linear) Datalog query. We then present the algorithms, which have the above transformation as their building block, that rewrite a query $Q \in (\mathsf{WARD} \cap \mathsf{PWL}, \mathsf{CQ})$ (resp., $Q \in (\mathsf{WARD}, \mathsf{CQ})$) into an equivalent query $Q' \in \mathsf{PWL\text{-}DATALOG}$ (resp., $Q' \in \mathsf{DATALOG}$).

From Proof Trees to Datalog. Consider a proof tree $\mathcal{P}$ of a CQ $q(\bar{x})$ w.r.t. a set Σ of TGDs. A node of $\mathcal{P}$, together with its children, is converted into a single-head full TGD that is added to (an initially empty) set $\Sigma_{\mathcal{P}}$. Assume that the node v has the children $u_1, \ldots, u_k$ in $\mathcal{P}$, where v is labeled by $p_0(\bar{x}_0)$ and, for $i \in [k]$, u_i is labeled by the CQ $p_i(\bar{x}_i)$ with $\bar{x}_0 \subseteq \bar{x}_i$. We add to $\Sigma_{\mathcal{P}}$ the full TGD

$$C_{[p_1]}(\bar{x}_1), \ldots, C_{[p_k]}(\bar{x}_k) \rightarrow C_{[p_0]}(\bar{x}_0),$$

where $C_{[p_i]}$ is a new predicate, not occurring in $\mathsf{sch}(\Sigma)$, that corresponds to the CQ p_i, while $[p_i]$ refers to the *canonical representative* of p_i. The intention underlying such a canonical representative is the following: if p_i and p_j are the same up to variable renaming, then $[p_i] = [p_j]$. Note that there are several different ways to define the canonical representative of a CQ p, for example, by applying a canonical renaming that always rewrites CQs that are the same up to variable renaming into the same CQ. Here we simply assume a fixed mechanism that computes the canonical representative $[p]$ of a CQ p. We also add to $\Sigma_{\mathcal{P}}$ a full TGD

$$R(y_1, \ldots, y_m) \rightarrow C_{[p_R]}(y_1, \ldots, y_m)$$

for each m-ary predicate $R \in \mathsf{edb}(\Sigma)$, where $p_R(y_1, \ldots, y_m)$ is the atomic query

$$Q(y_1, \ldots, y_m) \leftarrow R(y_1, \ldots, y_m).$$

This concludes the definition of $\Sigma_{\mathcal{P}}$. Observe that since in $\mathcal{P}$ we may have several node labels (i.e., CQs) that are the same up to variable renaming, the set $\Sigma_{\mathcal{P}}$ is, in general, recursive. Furthermore, if $\mathcal{P}$ is linear, then the employed recursion is piece-wise linear, i.e., $\Sigma_{\mathcal{P}} \in \mathsf{FULL}_1 \cap \mathsf{PWL}$. We finally define $Q_{\mathcal{P}}$ as the query $(\Sigma_{\mathcal{P}}, q_{\mathcal{P}}(\bar{x}))$, where $q_{\mathcal{P}}(\bar{x})$ is the atomic query

$$Q(\bar{x}) \leftarrow C_{[q]}(\bar{x}).$$

Here is a simple example that illustrates the above construction.

Example 6.5. Consider the set Σ of TGDs given in Example 4.5, which we repeat here:

$$R(x) \rightarrow \exists y\, T(y, x) \qquad T(x, y), S(y, z) \rightarrow T(x, z) \qquad T(x, y), P(y) \rightarrow G().$$

Consider also the CQ that simply asks whether $G()$ is entailed, i.e., the CQ

$$Q \leftarrow G(),$$

which has also been considered in Example 4.5. Let $\mathcal{P}$ be the proof tree of q w.r.t. Σ that encodes the branch $q_0, \ldots, q_n$ of the unfolding of q with Σ for $n = 4$. In other words, $\mathcal{P}$ is the tree consisting of the trees T_1, T_{n-2} and T_{n-1} depicted in Figure 1. According to the above construction, $\mathcal{P}$ is converted into the following set $\Sigma_{\mathcal{P}}$ of single-head full TGDs; note that we do not keep the TGDs that are already present (up to variable renaming). For brevity, we adopt the following naming convention: $p_{i,j}$ is the CQ that labels the j-th node (from left-to-right) of the i-th level of $\mathcal{P}$. If the i-th level has only one node, we simply write p_i. With this naming convention, the root is labeled with p_0, its

child with p_1, etc. The set of full TGDs $\Sigma_{\mathcal{P}}$ obtained from $\mathcal{P}$ consists of:

$$\begin{aligned}
C_{[p_1]} &\rightarrow C_{[p_0]}() \\
C_{[p_2]}(y^1) &\rightarrow C_{[p_1]}() \\
C_{[p_{3,1}]}(y^1), C_{[p_{3,2}]}(y^1) &\rightarrow C_{[p_2]}(y^1) \\
C_{[p_4]}(y^1) &\rightarrow C_{[p_{3,1}]}(y^1) \\
C_{[p_5]}(y^1, y^2) &\rightarrow C_{[p_4]}(y^1) \\
C_{[p_{3,1}]}(y^2), C_{[p_{6,2}]}(y^1, y^2) &\rightarrow C_{[p_5]}(y^1, y^2) \\
C_{[p_{10}]}(y^3) &\rightarrow C_{[p_{3,1}]}(y^3) \\
P(y^1) &\rightarrow C_{[p_{3,2}]}(y^1) \\
S(y^2, y^1) &\rightarrow C_{[p_{6,2}]}(y^1, y^2) \\
R(y^3) &\rightarrow C_{[p_{10}]}(y^3).
\end{aligned}$$

Finally, the desired Datalog query $Q_{\mathcal{P}}$ is defined as $(\Sigma_{\mathcal{P}}, Q \leftarrow C_{[p_0]}())$. ∎

It is not difficult to verify that the following holds – it actually follows by construction – which forms the building block of the rewriting algorithms that come next:

Lemma 6.6. *Consider a proof tree $\mathcal{P}$ of a CQ $q(\bar{x})$ w.r.t. a set Σ TGDs. Then:*
(1) $Q_{\mathcal{P}} \in$ DATALOG; furthermore, if $\mathcal{P}$ is linear, then $Q_{\mathcal{P}} \in$ PWL-DATALOG.
(2) For every database D over $\mathsf{edb}(\Sigma)$, $\mathcal{P}(D) = Q_{\mathcal{P}}(D)$.

The Rewriting Algorithm for WARD∩PWL**.** By exploiting the construction described above, and in particular Lemma 6.6, we can now rewrite every query $Q = (\Sigma, q) \in (\mathsf{WARD} \cap \mathsf{PWL}, \mathsf{CQ})$ into an equivalent query $Q' = (\Sigma', q')$ that falls in PWL-DATALOG. The idea is essentially to convert each linear proof tree $\mathcal{P}$ of q w.r.t. Σ such that $\mathsf{nwd}(\mathcal{P}) \leq f_{\mathsf{WARD}\cap\mathsf{PWL}}(q, \Sigma)$ (by Theorem 4.12, it suffices to consider only linear proof trees of node-width at most $f_{\mathsf{WARD}\cap\mathsf{PWL}}(q, \Sigma)$) into the piece-wise linear Datalog query $Q_{\mathcal{P}}$, and then take the union of all those queries. This is done via Algorithm 3, which we now describe. After converting the given set of TGDs into a set of single-head TGDs in level-wise normal form, we then perform an initialization step. The variable *ctr* is counting the number of resolution steps, which will then be used to rename the variables occurring in TGDs in order to avoid undesirable clutter during resolution. The set *pending* is collecting the CQs (i.e., nodes of proof trees) that are waiting to be processed by the current iteration of the algorithm (see the outer for-loop inside the repeat-until loop). The set *new* is collecting the new nodes that are generated during the current iteration, which will be then processed by the subsequent iteration. Finally, the set *explored* is collecting all the nodes that have been already processed, which allows us to identify whether a generated node is really new. The algorithm first computes all the possible root nodes $p(\bar{y})$ of proof trees by considering all the possible partitions π of $\bar{x}$, and then generates the output TGDs $C_{[p]}(\bar{y}) \rightarrow \mathsf{Ans}(\bar{y})$ that are added to Σ'; recall that $[p]$ is the canonical representation of p. Then, the algorithm exhaustively applies resolution steps, specialization steps, and decomposition steps, until there are no pending nodes, while at each step the corresponding TGD is constructed and added to Σ'. Notice that during a resolution step only MGCUs that use a substitution that is the identity on the output variables of the CQ under consideration are considered; those MGCUs are collected in the set $\mathsf{mgcu}_{\mathsf{ido}}(\cdot, \cdot)$. Notice also that during the actual resolution step via an MGCU U, the algorithm first renames the variables of the TGD σ under consideration by exploiting the current value of *ctr*; this is indicated by the notation $p[\sigma_{ctr}, U_{ctr}]$. Let us also stress that during the decomposition steps only *intensional-extensional decompositions* are considered, that is, decompositions where at most

Algorithm 3: The rewriting algorithm for WARD ∩ PWL

Input: A query $Q = (\Sigma, q(\bar{x})) \in (\text{WARD} \cap \text{PWL}, \text{CQ})$
Output: A query $Q' = (\Sigma', q'(\bar{x})) \in (\text{FULL}_1 \cap \text{PWL}, \text{CQ})$

$\Sigma := \mathsf{N}_{\mathsf{lw}}(\mathsf{N}_{\mathsf{sh}}(\Sigma))$
$ctr := 0$; $\Sigma' := \emptyset$; $pending := \emptyset$; $new := \emptyset$; $explored := \emptyset$
foreach partition π of $\bar{x}$ **do**
  $p(\bar{y}) := Q(\text{eq}_\pi(\bar{x})) \leftarrow \text{eq}_\pi(\alpha_1, \ldots, \alpha_n)$ with $\text{atoms}(q) = \{\alpha_1, \ldots, \alpha_n\}$
  $\Sigma' := \Sigma' \cup \{C_{[p]}(\bar{y}) \rightarrow \text{Ans}(\bar{y})\}$
  $pending := pending \cup \{p\}$
repeat
  foreach $p(\bar{y}) \in pending$ **do**
    /* resolution steps */
    foreach $\sigma \in \Sigma$ **do**
      foreach $U \in \text{mgcu}_{\text{ido}}(p, \sigma)$ **do**
        $ctr := ctr + 1$
        $p' := p[\sigma_{ctr}, U_{ctr}]$
        if $|p'| \leq f_{\text{WARD}\cap\text{PWL}}(q, \Sigma)$ **then**
          $\Sigma' := \Sigma' \cup \{C_{[p']}(\bar{y}) \rightarrow C_{[p]}(\bar{y})\}$
          if there is no $p'' \in explored$ such that $[p'] = [p'']$ **then**
            $new := new \cup \{p'\}$
    /* specialization steps */
    foreach specialization $p'(\bar{z})$ of $p(\bar{y})$ **do**
      $\Sigma' := \Sigma' \cup \{C_{[p']}(\bar{z}) \rightarrow C_{[p]}(\bar{y})\}$
      if there is no $p'' \in explored$ such that $[p'] = [p'']$ **then**
        $new := new \cup \{p'\}$
    /* decomposition steps */
    foreach extensional-intensional decomposition $\{p_1(\bar{z}_1), \ldots, p_k(\bar{z}_k)\}$ of $p(\bar{y})$ **do**
      $\Sigma' := \Sigma' \cup \{C_{[p_1]}(\bar{z}_1), \ldots, C_{[p_k]}(\bar{z}_k) \rightarrow C_{[p]}(\bar{y})\}$
      foreach $p' \in \{p_1, \ldots, p_k\}$ **do**
        if there is no $p'' \in explored$ such that $[p'] = [p'']$ **then**
          $new := new \cup \{p'\}$
    $explored := explored \cup \{p\}$
  $pending := new$; $new := \emptyset$
until $pending = \emptyset$;
foreach m-ary predicate $R \in \text{edb}(\Sigma)$ **do**
  $p_R(\bar{y}) := Q(y_1, \ldots, y_m) \leftarrow R(y_1, \ldots, y_m)$
  $\Sigma' := \Sigma' \cup \{R(\bar{y}) \rightarrow C_{[p_R]}(\bar{y})\}$
$q'(\bar{x}) := Q(x_1, \ldots, x_n) \leftarrow \text{Ans}(x_1, \ldots, x_n)$
$Q' := (\Sigma', q'(\bar{x}))$
return Q'

one CQ mentions intensional predicates of Σ; this ensures that only linear proof trees are explored. If there are no pending CQs, which means that all the possible linear proof trees of q w.r.t. Σ with node-width at most $f_{\text{WARD}\cap\text{PWL}}(q, \Sigma)$ have been converted into single-head full TGDs, the algorithm proceeds to generate the input TGDs that will be triggered by the given database; these are the TGDs of the form $R(\bar{y}) \rightarrow C_{[p_R]}(\bar{y})$ for $R \in \text{edb}(\Sigma)$. Finally, the desired query $Q' = (\Sigma', q'(\bar{x}))$ is

generated. The next technical lemma shows that Algorithm 3 effectively rewrites a query from (WARD ∩ PWL, CQ) into a query from PWL-DATALOG.

Lemma 6.7. *Consider a query $Q = (\Sigma, q(\bar{x})) \in (\mathsf{WARD} \cap \mathsf{PWL}, \mathsf{CQ})$, and let $Q' = (\Sigma', q')$ be the output of Algorithm 3 on input Q. The following hold:*

(1) Σ' is finite and falls in $\mathsf{FULL}_1 \cap \mathsf{PWL}$; hence, $Q' \in$ PWL-DATALOG.
(2) For every database D over $\mathsf{edb}(\Sigma)$, $Q(D) = Q'(D)$.

Proof. Concerning item (1), it follows by construction, since we explore only linear proof trees that is guaranteed by the fact that only intensional-extensional decompositions are considered, that Σ' is a piece-wise linear set of single-head full TGDs. The fact that Σ' is finite follows from the fact that only the canonical representation of the CQs that label the nodes of a linear proof tree of q w.r.t. Σ are considered, while the number of atoms occurring in those CQs is bounded by $f_{\mathsf{WARD}\cap\mathsf{PWL}}(q, \Sigma)$. In other words, only finitely many (in fact, exponentially many) CQs are actually explored during the execution of the algorithm, which in turn implies the finiteness of Σ'.

Concerning item (2), assume first that $\bar{c} \in Q(D)$ for some tuple $\bar{c} \in \mathsf{dom}(D)^{|\bar{x}|}$. By Theorem 4.12, there exists a linear proof tree $\mathcal{P}$ of q w.r.t. Σ with $\mathsf{nwd}(\mathcal{P}) \leq f_{\mathsf{WARD}\cap\mathsf{PWL}}(q, \Sigma)$ such that $\bar{c} \in \mathcal{P}(D)$. By construction and Lemma 6.6, there exists a set $\Sigma_{\mathcal{P}} \subseteq \Sigma'$ such that $\mathcal{P}(D) = Q_{\mathcal{P}}(D)$ with $Q_{\mathcal{P}} = (\Sigma_{\mathcal{P}}, q')$. Therefore, $\bar{c} \in Q_{\mathcal{P}}(D)$, which implies that $\bar{c} \in Q'(D)$. Conversely, assume that $\bar{c} \in Q'(D)$. By construction, there exists a set $\Sigma_{\mathcal{P}} \subseteq \Sigma'$ such that $\Sigma_{\mathcal{P}}$ is obtained after converting a linear proof tree $\mathcal{P}$ of q w.r.t. Σ with $\mathsf{nwd}(\mathcal{P}) \leq f_{\mathsf{WARD}\cap\mathsf{PWL}}(q, \Sigma)$ into full TGDs, and $\bar{c} \in Q_{\mathcal{P}}(D)$ with $Q_{\mathcal{P}} = (\Sigma_{\mathcal{P}}, q')$. By Lemma 6.6, we get that $\bar{c} \in \mathcal{P}(D)$. Therefore, by Theorem 4.12, we conclude that $\bar{c} \in Q(D)$. □

Item (1) of Lemma 6.4 is an immediate consequence of Lemma 6.7.

The Rewriting Algorithm for WARD. The rewriting of a query $Q = (\Sigma, q) \in (\mathsf{WARD}, \mathsf{CQ})$ into an equivalent query $Q' = (\Sigma', q')$ that falls in DATALOG is done in a similar way as for piece-wise linear warded sets of TGDs, i.e., we convert each proof tree (not necessarily linear) $\mathcal{P}$ of q w.r.t. Σ such that $\mathsf{nwd}(\mathcal{P}) \leq f_{\mathsf{WARD}}(q, \Sigma)$ (by Theorem 4.13, it suffices to consider only proof trees of node-width at most $f_{\mathsf{WARD}}(q, \Sigma)$) into the Datalog query $Q_{\mathcal{P}}$, and then take the union of all those queries. This is done via an adapted version of Algorithm 3; here are the only differences:

(1) The given set Σ of TGDs is converted into single-head normal form, but not into level-wise normal form, i.e., the first line is replaced by $\Sigma := \mathsf{N}_{\mathsf{sh}}(\Sigma)$.
(2) The node-width of the explored proof trees is at most $f_{\mathsf{WARD}}(q, \Sigma)$, i.e., the condition of the first if-statement in the for-loop that performs resolution steps is replaced by $|p'| \leq f_{\mathsf{WARD}}(q, \Sigma)$.
(3) Proof trees that are not linear are also explored, i.e., the for-loop that performs decomposition steps considers all the decompositions (not only the extensional-intensional ones) of $p(\bar{y})$.

Having the above adapted version of Algorithm 3 in place, we immediately get a result for queries from (WARD, CQ) analogous to Lemma 6.7, which in turn implies item (2) of Lemma 6.4.

6.2 Program Expressive Power

The *expressive power* of a set Σ of TGDs, denoted $\mathsf{ep}(\Sigma)$, is the set of triples $(D, q(\bar{x}), \bar{c})$, where D is a database over $\mathsf{edb}(\Sigma)$, $q(\bar{x})$ is a CQ over $\mathsf{sch}(\Sigma)$, and $\bar{c} \in \mathsf{dom}(D)^{|\bar{x}|}$, such that $\bar{c} \in \mathsf{cert}(q, D, \Sigma)$. The *program expressive power* of a query language $(\mathsf{C}, \mathsf{CQ})$, where C is a class of TGDs, is defined as

$$\mathsf{pep}(\mathsf{C}, \mathsf{CQ}) \;=\; \{\mathsf{ep}(\Sigma) \mid \Sigma \in \mathsf{C}\}.$$

Given two query languages $\mathsf{Q}_1, \mathsf{Q}_2$, we say that Q_2 is *more expressive (w.r.t. program expressive power)* than Q_1, written $\mathsf{Q}_1 \leq_{\mathsf{pep}} \mathsf{Q}_2$, if $\mathsf{pep}(\mathsf{Q}_1) \subseteq \mathsf{pep}(\mathsf{Q}_2)$. Moreover, we say that Q_2 is *strictly more expressive (w.r.t. the program expressive power)* than Q_2, written $\mathsf{Q}_1 <_{\mathsf{pep}} \mathsf{Q}_2$, if $\mathsf{Q}_1 \leq_{\mathsf{pep}} \mathsf{Q}_2$ and

$\mathrm{Q}_2 \not\leq_{\mathrm{pep}} \mathrm{Q}_1$. It is easy to show a lemma analogous to Lemma 6.2, which reveals the essence of the program expressive power. For brevity, given two classes of TGDs C_1 and C_2, we write $\mathrm{C}_1 \preceq \mathrm{C}_2$ if, for every $\Sigma \in \mathrm{C}_1$, there exists $\Sigma' \in \mathrm{C}_2$ such that, for every D over $\mathrm{edb}(\Sigma)$, and CQ q over $\mathrm{sch}(\Sigma)$, $Q(D) = Q'(D)$, where $Q = (\Sigma, q)$ and $Q' = (\Sigma', q)$.

Lemma 6.8. *For two query languages* $\mathrm{Q}_1 = (\mathrm{C}_1, \mathrm{CQ})$ *and* $\mathrm{Q}_2 = (\mathrm{C}_2, \mathrm{CQ})$, $\mathrm{Q}_1 \leq_{\mathrm{pep}} \mathrm{Q}_2$ *iff* $\mathrm{C}_1 \preceq \mathrm{C}_2$.

We are now ready to study the expressiveness (w.r.t. the program expressive power) of (WARD ∩ PWL, CQ) and (WARD, CQ) relative to Datalog. In particular:

Theorem 6.9. *The following statements hold:*

(1) PWL-DATALOG $<_{\mathrm{pep}}$ (WARD ∩ PWL, CQ).
(2) DATALOG $<_{\mathrm{pep}}$ (WARD, CQ).

Proof. We proceed to show (1); the second statement is shown in the same way. First, observe that PWL-DATALOG $\leq_{\mathrm{pep}}$ (WARD∩PWL, CQ) since, by definition, $\mathsf{FULL}_1 \cap \mathsf{PWL} \subseteq \mathsf{WARD} \cap \mathsf{PWL}$, and thus $\mathsf{FULL}_1 \cap \mathsf{PWL} \preceq \mathsf{WARD} \cap \mathsf{PWL}$; the claim follows by Lemma 6.8. It remains to show that (WARD ∩ PWL, CQ) $\not\leq_{\mathrm{pep}}$ PWL-DATALOG. By Lemma 6.8, this boils down to showing that $\mathsf{WARD} \cap \mathsf{PWL} \not\preceq \mathsf{FULL}_1 \cap \mathsf{PWL}$. By contradiction, assume the opposite. Consider the set of TGDs $\Sigma = \{P(x) \rightarrow \exists y\, R(x, y)\}$, which is clearly in WARD ∩ PWL, the database $D = \{P(c)\}$, and the CQs

$$q_1 \;=\; Q \leftarrow R(x, y) \qquad \text{and} \qquad q_2 \;=\; Q \leftarrow R(x, y), P(y).$$

By hypothesis, there is a set of TGDs $\Sigma' \in \mathsf{FULL}_1 \cap \mathsf{PWL}$ such that $Q_1(D) = Q_1'(D)$ and $Q_2(D) = Q_2'(D)$, where $Q_i = (\Sigma, q_i)$ and $Q_i' = (\Sigma', q_i)$, for $i \in \{1, 2\}$. Clearly, $Q_1(D) \neq \emptyset$ and $Q_2(D) = \emptyset$, which implies that $Q_1'(D) \neq \emptyset$ and $Q_2'(D) = \emptyset$. Observe now that $Q_1'(D) \neq \emptyset$ implies $Q_2'(D) \neq \emptyset$. Indeed, if $Q_1'(D) \neq \emptyset$, then necessarily there exists an atom of the form $R(c, c)$ in $\mathsf{chase}(D, \Sigma')$ since $\mathsf{dom}(\mathsf{chase}(D, \Sigma')) = \mathsf{dom}(D)$; the latter holds due to the fact that Σ' is constant-free and does not use existentially quantified variables. Therefore, due to the atoms $P(c)$ and $R(c, c)$ in $\mathsf{chase}(D, \Sigma')$, $Q_2'(D) \neq \emptyset$. But this contradicts the fact that $Q_2'(D) = \emptyset$, and the claim follows. □

7 THE PRACTICAL EFFECT OF PIECE-WISE LINEARITY ON VADALOG

After studying in depth the theoretical aspects of piece-wise linearity, we now proceed to give preliminary experimental evidence for its practical effect on the Vadalog system, which has been originally designed for warded sets of TGDs. But let us first give a brief description of the Vadalog system, which will then allow us to give a relatively self-contained overview for what follows. The interested reader can find additional technical details concerning the Vadalog system in [9], but all the required information for understanding the rest of the section will be presented here.

7.1 The Vadalog System: A Brief Description

Our analysis on how piece-wise linearity affects the performance of the Vadalog system heavily relies on how piece-wise linearity affects the way that existential quantifiers interact with recursion. A crucial issue when combining existential quantification and recursion in any TGD language is that of non-termination. In the Vadalog system, a separate subsystem has been designed that is responsible for termination control. To understand the effect that piece-wise linearity has on Vadalog and this subsystem, we first have to describe how the system executes warded (non necessarily piece-wise linear) sets of TGDs in general.

Reasoning Query Plan. Unlike many rule-based reasoners, and similar to many database systems, the Vadalog system translates a set of TGDs into a pipeline-style network of operators, called the *reasoning query plan*. An abstract example of such a plan is shown in Figure 2. We first focus on

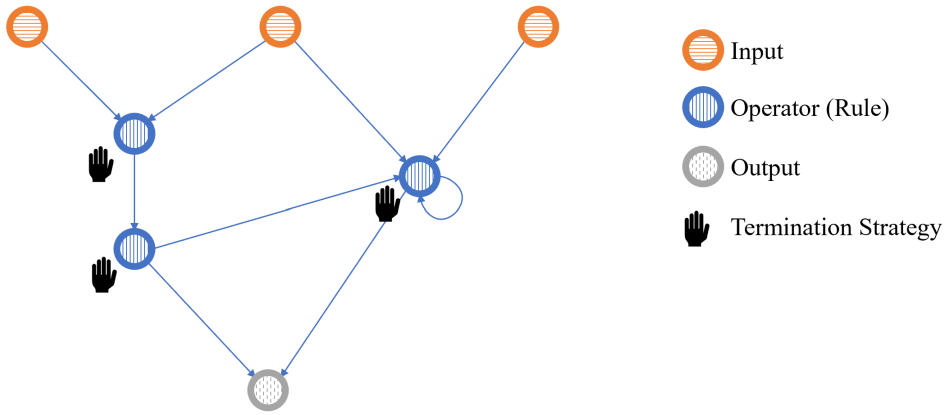


Fig. 2. Reasoning query plan.

the circular nodes representing the nodes of our reasoning query plan. At the top of the figure are the input nodes (shown in orange, horizontally shaded). An input node represents a relation, coming from a traditional relational database, an RDF store, or any other data source able to expose a stream of relational data. At the bottom we have an output node (shown in grey, vertically-dashed shaded); note that, in general, there may be more than one output node. Such output nodes represent relations the user wishes to consume as output. All the other nodes represent TGDs in their original form, or as the result of the optimizer of the system rewriting them. As in typical query plans, an edge indicates the output of one node being the input of another node.

Termination Control. As said above, in the Vadalog system, a separate subsystem has been designed that is responsible for termination control. In particular, termination strategies are associated with some of the nodes in a reasoning query plan. These strategies are indicated by a "hand" symbol in Figure 2. Considering the right-most node in the figure with a termination strategy, we see that this node is also recursive, which is indicated via the looping edge. In such a case, namely TGDs that are also part of a recursion, termination strategies ensure that the computation terminates, which is generally not the case. Furthermore, termination strategies try to terminate the execution as early as possible by exploiting the wardedness condition while ensuring correctness (i.e., not losing any output tuples, or producing incorrect tuples). In other words, termination strategies are also responsible for pruning tuples that are unnecessary for the output to be produced.

For the purpose of termination control, the system builds some guide termination structures: the linear forest, the warded forest, and the lifted linear forest; for details see [9]. Although those termination structures are central on how the Vadalog system works, for our purposes here it suffices to know that they exist, and drop tuples (i.e., block execution) at certain points. Let us remark again that adding or removing a termination strategy to a node only affects the termination behaviour, but not the correctness of the system, which is guaranteed by construction.

Modes of Termination Control. In Figure 2, the only potential source for non-termination is the right-most node marked with the "hand" symbol (i.e., an attached termination strategy) that also has a loop attached to it. It should be clear that removing the termination strategy from the other two nodes will not affect termination since the node with the actual cycle still has a termination strategy attached to it. Yet, it is always possible to statically determine which of the termination strategies can be omitted, and which must remain. We call the setting of the Vadalog engine with

termination strategies attached to all nodes representing TGDs (i.e., not input and output nodes) the *standard* termination strategy. The one with all the termination strategies removed, apart from those that are really needed to ensure termination, is called the *pruned* termination strategy. Let us stress, however, that this is actually a trade-off, i.e., using a termination strategy requires time (including creating and querying the data structures of the termination strategies) while it potentially also saves time (by dropping tuples not really needed in the computation).

7.2 Experimental Scenario

For our experiment, we are going to consider a scenario from Section 6.3 of [9] built on top of DBpedia. The general setup of this scenario is on companies, company control and key persons stored in DBpedia. The scenario considers the `Company` entity in DBpedia mapped to facts of the form Company(c). The control relation is populated by the `dbo:parentCompany` relationship in DBpedia mapped to facts of the form Control(c, d), that is, c controls d if c is the parent company of d. The key persons relation is populated by the `dbo:keyPerson` relationship in DBpedia mapped to facts of the form KeyPerson(c, p) if p is a key person of company c. DBpedia contains information about approximately 67K companies and approximately 1.5M persons.

The scenario is on *persons with significant control* (PSCs), which are all persons of a company who directly or indirectly have significant control of a company. Every company has at least one PSC, either one provided by the data (as key persons) or an unknown person. Based on those PSCs, the *strong links* between companies are companies that share a certain number N of persons of significant control. We are going to consider the computationally hardest of the tasks considered in [9], namely the one where the goal is to compute all such strong links, and we are going to set $N = 1$ in order to produce the highest number of such strong links. Let us clarify though that the scenario in [9] allows for a broader variation (allowing a variable parameter N for strong links, computing the actual weight, i.e., the number of PSCs a strong link is based on, etc.), but also uses advanced Vadalog features that go beyond TGDs.

$$\text{KeyPerson}(x, u) \rightarrow \text{SignificantCtrl}(x, x, u) \tag{1}$$
$$\text{Company}(x) \rightarrow \exists u\, \text{SignificantCtrl}(x, x, u) \tag{2}$$
$$\text{Control}(x, y) \rightarrow \exists u\, \text{SignificantCtrl}(x, y, u) \tag{3}$$

$$\text{Control}(x, y), \text{SignificantCtrl}(y, z, u) \rightarrow \text{SignificantCtrl}(x, z, u) \tag{4}$$
$$\text{Control}(x, y), \text{SignificantCtrl}(x, x, u) \rightarrow \text{SignificantCtrl}(y, y, u) \tag{5}$$

$$\text{SignificantCtrl}(\_, x, u), \text{SignificantCtrl}(\_, y, u) \rightarrow \text{StrongLink}(x, y) \tag{6}$$

Fig. 3. The piece-wise linear warded set of TGDs Σ_{PWL}.

The piece-wise linear warded set of TGDs Σ_{PWL} expressing the above scenario is given in Figure 3. It uses one main recursive relation into which all relevant information is loaded, and from which all relevant information is extracted. The meaning of the main recursive relation SignificantCtrl(x, y, u) is that company x controls company y, and that company y has person of significant control u. The TGDs (1)-(3) are responsible for loading the source data into the main relation. TGD (4) is responsible for transitively closing company control (the first two arguments of SignificantCtrl).

TGD (5) is responsible for propagating persons of significant control (the last two arguments of SignificantCtrl). Finally, TGD (6) extracts the relevant information, i.e., all strong links.

$$\begin{aligned} \text{KeyPerson}(x,u) &\rightarrow \text{SignificantCtrl}(x,x,u) \\ \text{Company}(x) &\rightarrow \exists u\, \text{SignificantCtrl}(x,x,u) \\ \text{Control}(x,y) &\rightarrow \exists u\, \text{SignificantCtrl}(x,y,u) \end{aligned}$$

$$\text{SignificantCtrl}(x,y,v), \text{SignificantCtrl}(y,z,u) \rightarrow \text{SignificantCtrl}(x,z,u)$$
$$\text{SignificantCtrl}(x,y,v), \text{SignificantCtrl}(x,x,u) \rightarrow \text{SignificantCtrl}(y,y,u)$$

$$\text{SignificantCtrl}(v,x,u), \text{SignificantCtrl}(w,y,u) \rightarrow \text{StrongLink}(x,y)$$

Fig. 4. The warded set of TGDs Σ_{WARD}.

With the aim of investigating the effect of piece-wise linearity on the Vadalog system, we are also considering the (non-piece-wise linear) warded set of TGDs Σ_{WARD}, depicted in Figure 4, which is an equivalent variant of Σ_{PWL}, obtained from Σ_{PWL} by simply replacing the atom $\text{Control}(x,y)$ in TGDs (4) and (5) with the atom $\text{SignificantCtrl}(x,y,v)$.

7.3 Experimental Evaluation and Discussion

In our experiment, to see the effect of piece-wise linearity on the interaction between recursion and existential quantification (the two key ingredients for non-termination), we will consider the scenario described above. For the test set, we are going to use the same setting as in [9], that is, a constant set of 1.5 million persons and 1K, 10K, 25K, 50K and 67K companies. We note that the complexity of the computation is not rooted in the far larger set of 1.5 million persons, but the interconnections of the companies. DBpedia contains 67K such companies, and the smaller subsets have been obtained by random sampling. The experiment has been performed on a machine with

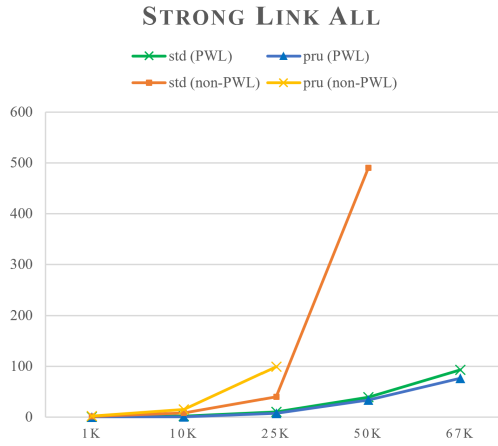


Fig. 5. PWL vs. non-PWL. Plotted are times in seconds vertically and the number of companies horizontally.

an Intel Core i7-8650U CPU on Oracle JDK 14 on top of a Windows 10 version 1909 host. The JVM was allocated a maximum of 6GB of memory.

In Figure 5 we see the performance of the Vadalog system in seconds plotted on the vertical axis, and the number of companies plotted on the horizontal axis. The four series represent:

- the piece-wise linear set of TGDs Σ_{PWL} with standard termination strategy (std PWL),
- the piece-wise linear set of TGDs Σ_{PWL} with pruned termination strategy (pru PWL),
- the warded set of TGDs Σ_{WARD} with standard termination strategy (std non-PWL), and
- the warded set of TGDs Σ_{WARD} with pruned termination strategy (pru non-PWL).

The experiment was executed with a timeout of 600s (10 minutes). There are three entries hitting this timeout, the (std non-PWL) configuration with 67K companies, and the (pru non-PWL) configuration with 50K and 67K companies.

Discussion. We first observe that the Vadalog system performs better with the set of TGDs Σ_{PWL} as input. Furthermore, we see that for Σ_{PWL} the pruned termination strategy (i.e., the one removing termination strategies from nodes where this can be done) has a slight positive effect compared to the standard termination strategy. This is what we usually expect, as the overhead of unnecessary termination strategy nodes is typically higher than the performance gained by the tuples dropped by them. Interestingly, this trend reverts when we consider as input the warded set of TGDs Σ_{WARD}. Indeed, the performance of the pruned termination strategy is much worse. The reason for this, which reveals the usefulness of piece-wise linear recursion, is that non-piece-wise-linear recursion generates duplicates. The standard termination strategy, having termination strategy nodes wherever possible, will prune those unnecessary tuples as early as possible. The pruned termination strategy will save this effort, and thus gain minimal performance, but will incur a huge performance penalty by having to process all those unnecessary duplicate tuples.

8 CONCLUSIONS AND FUTURE WORK

We have seen that restricting the recursion allowed by wardedness to piece-wise linear recursion leads to a formalism that provides a convenient syntax for expressing useful recursive statements, and at the same time achieves space-efficiency, in particular, NLogSpace data complexity.

Although the theoretical aspects of piece-wise linearity (i.e., complexity and expressive power) are rather well-understood, the practical side of this line of research is still at a very preliminary stage. We have provided initial experimental evidence for the practical effect of piece-wise linearity on the Vadalog system, and, in particular, on the subsystem that is responsible for termination control, but this is only a glimpse. Our long-term plan is to properly implement and experimentally evaluate piece-wise linearity as part of the Vadalog system. In fact, an optimized parallel implementation is currently under development. This relies on the fact that reasoning under piece-wise linear warded sets of TGDs is principally parallelizable, since NLogSpace is contained in the class NC_2 of highly parallelizable problems, unlike warded sets of TGDs for which reasoning is PTime-hard, and thus difficult to parallelize effectively. We are also planning to extend the benchmark suite iWarded for warded sets of TGDs to cover also piece-wise linear warded sets of TGDs in more detail. This will be crucial for the experimental evaluation of the parallel implementation mentioned above.

REFERENCES

[1] Foto N. Afrati, Manolis Gergatsoulis, and Francesca Toni. 2003. Linearisability on datalog programs. *Theor. Comput. Sci.* 308, 1-3 (2003), 199–226.

[2] Marcelo Arenas, Georg Gottlob, and Andreas Pieris. 2014. Expressive languages for querying the semantic web. In *PODS*. 14–26.

[3] Marcelo Arenas, Georg Gottlob, and Andreas Pieris. 2018. Expressive Languages for Querying the Semantic Web. *ACM Trans. Database Syst.* 43, 3 (2018), 13:1–13:45.

[4] Patricia C. Arocena, Boris Glavic, Radu Ciucanu, and Renée J. Miller. 2015. The iBench Integration Metadata Generator. *PVLDB* 9, 3 (2015), 108–119.
[5] Jean-François Baget, Marie-Laure Mugnier, Sebastian Rudolph, and Michaël Thomazo. 2011. Walking the Complexity Lines for Generalized Guarded Existential Rules. In *IJCAI*. 712–717.
[6] Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, and Eric Salvat. 2011. On rules with existential variables: Walking the decidability line. *Artif. Intell.* 175, 9-10 (2011), 1620–1654.
[7] Catriel Beeri and Moshe Y. Vardi. 1981. The Implication Problem for Data Dependencies. In *ICALP*. 73–85.
[8] Luigi Bellomarini, Georg Gottlob, Andreas Pieris, and Emanuel Sallinger. 2017. Swift Logic for Big Data and Knowledge Graphs. In *IJCAI*. 2–10.
[9] Luigi Bellomarini, Emanuel Sallinger, and Georg Gottlob. 2018. The Vadalog System: Datalog-based Reasoning for Knowledge Graphs. *PVLDB* 11, 9 (2018), 975–987.
[10] Michael Benedikt, George Konstantinidis, Giansalvatore Mecca, Boris Motik, Paolo Papotti, Donatello Santoro, and Efthymia Tsamoura. 2017. Benchmarking the Chase. In *PODS*. 37–52.
[11] Gerald Berger, Georg Gottlob, Andreas Pieris, and Emanuel Sallinger. 2019. The Space-Efficient Core of Vadalog. In *PODS*. 270–284.
[12] Meghyn Bienvenu, Balder ten Cate, Carsten Lutz, and Frank Wolter. 2014. Ontology-Based Data Access: A Study through Disjunctive Datalog, CSP, and MMSNP. *ACM Trans. Database Syst.* 39, 4 (2014), 33:1–33:44.
[13] Peter Van Emde Boas. 1997. The Convenience of Tilings. In *Complexity, Logic, and Recursion Theory*. 331–363.
[14] Andrea Calì, Georg Gottlob, and Michael Kifer. 2013. Taming the Infinite Chase: Query Answering under Expressive Relational Constraints. *J. Artif. Intell. Res.* 48 (2013), 115–174.
[15] Andrea Calì, Georg Gottlob, Thomas Lukasiewicz, Bruno Marnette, and Andreas Pieris. 2010. Datalog+/-: A Family of Logical Knowledge Representation and Query Languages for New Applications. In *LICS*. 228–242.
[16] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. 2005. Data exchange: semantics and query answering. *Theor. Comput. Sci.* 336, 1 (2005), 89–124.
[17] Georg Gottlob, Giorgio Orsi, and Andreas Pieris. 2014. Query Rewriting and Optimization for Ontological Databases. *ACM Trans. Database Syst.* 39, 3 (2014), 25:1–25:46.
[18] Georg Gottlob and Andreas Pieris. 2015. Beyond SPARQL under OWL 2 QL Entailment Regime: Rules to the Rescue. In *IJCAI*. 2999–3007.
[19] David S. Johnson and Anthony C. Klug. 1984. Testing Containment of Conjunctive Queries under Functional and Inclusion Dependencies. *J. Comput. Syst. Sci.* 28, 1 (1984), 167–189.
[20] Mélanie König, Michel Leclère, Marie-Laure Mugnier, and Michaël Thomazo. 2015. Sound, complete and minimal UCQ-rewriting for existential rules. *Semantic Web* 6, 5 (2015), 451–475.
[21] Roman Kontchakov, Martin Rezk, Mariano Rodriguez-Muro, Guohui Xiao, and Michael Zakharyaschev. 2014. Answering SPARQL Queries over Databases under OWL 2 QL Entailment Regime. In *ISWC*. 552–567.
[22] David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. 1979. Testing Implications of Data Dependencies. *ACM Trans. Database Syst.* 4, 4 (1979), 455–469.
[23] Jeffrey F. Naughton. 1986. Data Independent Recursion in Deductive Databases. In *PODS*. 267–279.
[24] Jeffrey F. Naughton and Yehoshua Sagiv. 1987. A Decidable Class of Bounded Recursions. In *PODS*. 227–236.

A PROOF OF THEOREM 4.9

Recall that a branch of the unfolding of q with Σ is a sequence of CQs $(q_i)_{i\in\{0,\ldots,n\}}$, where $q = q_0$, while, for each $i \in [n]$, q_i is a σ-resolvent of q_{i-1} for some $\sigma \in \Sigma$. It is not difficult to verify that the following statements are equivalent:

- There is a proof tree $\mathcal{P}$ of q w.r.t. Σ such that $\bar{c} \in \mathcal{P}(D)$.
- There is a branch $(q_i)_{i\in\{0,\ldots,n\}}$, for $n \geq 0$, of the unfolding of q with Σ such that $\bar{c} \in q_n(D)$.

Thus, to establish Theorem 4.9, it suffices to show that the following statements are equivalent:

(1) $\bar{c} \in \mathsf{cert}(q, D, \Sigma)$.
(2) There is a branch $(q_i)_{i\in\{0,\ldots,n\}}$, for $n \geq 0$, of the unfolding of q with Σ such that $\bar{c} \in q_n(D)$.

A.1 Proof of (1) ⇒ (2)

We first establish the following auxiliary lemma:

Lemma A.1. *Consider a CQ $q(\bar{x})$, and a prefix $(I_i)_{i\in\{0,\ldots,m\}}$, where $m \geq 0$, of a chase sequence for D under Σ with $I_i\langle\sigma_i, h_i\rangle I_{i+1}$, for each $i \in \{0,\ldots,m-1\}$, such that $\bar{c} \in q(I_m)$. Then, there exists a sequence $(q_i)_{i\in\{0,\ldots,m\}}$ of CQs such that the following hold:*

- *$q_0 = q$,*
- *$q_i = q_{i-1}$ or q_i is a σ_{m-i}-resolvent of q_{i-1}, for each $i \in \{1,\ldots,m\}$, and*
- *$\bar{c} \in q_i(I_{m-i})$, for each $i \in \{0,\ldots,m\}$.*

Proof. We proceed by induction on the length of the chase sequence.

Base Case. The statement holds trivially since $\bar{c} \in q(I_0)$. In other words, the desired sequence of CQs consists only of q.

Induction Step. Assume that there is a prefix $(I_i)_{i\in\{0,\ldots,m+1\}}$ of a chase sequence for D under Σ with $I_i\langle\sigma_i, h_i\rangle I_{i+1}$ for each $i \in \{0,\ldots,m\}$ such that $\bar{c} \in q(I_{m+1})$. Clearly, there exists a homomorphism μ such that $\mu(\text{atoms}(q)) \subseteq I_{m+1}$ and $\mu(\bar{x}) = \bar{c}$. Let $H = h'_m(\text{head}(\sigma_m))$, where h'_m is an extension of h_m that maps the existentially quantified variables of σ_m to fresh nulls. It is clear that $H \subseteq I_{m+1}$. We proceed by considering the following two cases:

Case 1. Assume that $H \cap \mu(\text{atoms}(q)) = \emptyset$. This implies that $\bar{c} \in q(I_m)$. Therefore, by induction hypothesis, there exists a sequence of CQs $(q'_i)_{i\in\{0,\ldots,m\}}$, where $q'_0 = q$, that enjoys the desired properties. Thus, the claim follows due to the the sequence of CQs $q, q, q'_1, \ldots, q'_m$.

Case 2. The interesting case is when $H \cap \mu(\text{atoms}(q)) \neq \emptyset$. Let $S \subseteq \text{atoms}(q)$ such that $\mu(S) \subseteq H$, while $H \cap \mu(\text{atoms}(q) \setminus S) = \emptyset$. In other words, S is the maximal subset of $\text{atoms}(q)$ that is mapped to H via μ. Let $S' \subseteq \text{head}(\sigma_m)$ such that $h'_m(S') = \mu(S)$. It is easy to verify that (S, S', γ), where $\gamma = h'_m \cup \mu$, is a chunk unifier of q with σ_m; we assume, w.l.o.g., that σ_m and q do not share variables, and thus, γ is a well-defined substitution. Indeed, for every $x \in \text{var}(S') \cap \text{var}_\exists(\sigma_m)$, $\gamma(x)$ is not a constant since, by construction, $h'_m(x)$ is a null, and $\gamma(x) = \gamma(y)$ implies that y occurs in S and is not shared. By contradiction, assume that y is shared, which means that it occurs in $\text{atoms}(q) \setminus S$. Observe that, by definition of the set S, $\mu(\text{atoms}(q) \setminus S) \subseteq I_m$, and thus, $\mu(y) = \gamma(y)$ is a null occurring in I_m, which is a contradiction since $\mu(y)$ has been invented in H, which means that it occurs only in $I_{m+1} \setminus I_m$. Since (S, S', γ) is a chunk unifier of q with σ_m, there exists a most general one $(S, S', \hat{\gamma})$. We define $\hat{q}$ in such a way that $\text{atoms}(\hat{q}) = \hat{\gamma}((\text{atoms}(q) \setminus S) \cup \text{body}(\sigma_m))$ i.e., as a σ_m-resolvent of q, while its free variables are $\hat{\gamma}(\bar{x})$. We can show that $\bar{c} \in \hat{q}(I_m)$, i.e., there exists a homomorphism λ such that $\lambda(\text{atoms}(\hat{q})) \subseteq I_m$ and $\lambda(\hat{\gamma}(\bar{x})) = \bar{c}$. By definition of the MGU, $\gamma = \theta \circ \hat{\gamma}$ for some substitution θ. It is clear that θ maps $\text{atoms}(\hat{q})$ to I_m since $\gamma(\text{body}(\sigma_m)) \subseteq I_m$ and $\gamma(\text{atoms}(q) \setminus S) \subseteq I_m$. Moreover, $\theta(\hat{\gamma}(\bar{x})) = \gamma(\bar{x}) = \bar{c}$. Thus, $\bar{c} \in \hat{q}(I_m)$ as claimed above. By induction hypothesis, there exists a sequence of CQs $(q'_i)_{i\in\{0,\ldots,m\}}$, where $q'_0 = \hat{q}$, that enjoys the desired properties. Thus, the claim follows due to the the sequence of CQs $q, \hat{q}, q'_1, \ldots, q'_m$.

This completes the proof of Lemma A.1. □

We can now complete the proof of the statement (1) $\Rightarrow$ (2). By hypothesis, $\bar{c} \in \text{cert}(q, D, \Sigma)$. Thus, by Proposition 2.1, there exists a prefix $(I_i)_{i\in\{0,\ldots,m\}}$, where $m \geq 0$, such that $\bar{c} \in q(I_m)$. By Lemma A.1, there exists a sequence of CQs $(q_i)_{i\in\{0,\ldots,m\}}$, where $q_0 = q$, q_i is either q_{i-1} or a σ-resolvent of q_{i-1}, where $\sigma \in \Sigma$, for each $i \in \{1,\ldots,m\}$, and $\bar{c} \in q_m(D)$. However, strictly speaking, $(q_i)_{i\in\{0,\ldots,m\}}$ is not a branch of the unfolding of q with Σ due to the fact that some CQs are repeated. Indeed, there are indices $i \in \{1,\ldots,m\}$ such that q_i is not a resolvent of q_{i-1} but the same CQ q_{i-1}. We can easily convert $(q_i)_{i\in\{0,\ldots,m\}}$ into a proper branch of the unfolding of q with Σ of length $n \leq m$ by simply removing the repeated CQs from the sequence $(q_i)_{i\in\{0,\ldots,m\}}$.

A.2 Proof of (2) ⇒ (1)

We first establish the following auxiliary lemma:

Lemma A.2. *Consider a branch* $(q_i)_{i\in\{0,\ldots,n\}}$*, for* $n \geq 0$*, of the unfolding of* q *with* Σ*. For every* $i \in \{0,\ldots,n\}$*,* $\bar{c} \in \mathsf{cert}(q_i, D, \Sigma)$ *implies* $\bar{c} \in \mathsf{cert}(q, D, \Sigma)$*.*

Proof. We proceed by induction on $i \geq 0$.

Base Case. Clearly, $\bar{c} \in \mathsf{cert}(q_0, D, \Sigma)$ implies $\bar{c} \in \mathsf{cert}(q, D, \Sigma)$ holds trivially since $q_0 = q$.

Induction Step. Suppose now that $\bar{c} \in \mathsf{cert}(q_i, D, \Sigma)$, for $i > 0$. To show that $\bar{c} \in \mathsf{cert}(q, D, \Sigma)$, by induction hypothesis, it suffices to show that $\bar{c} \in \mathsf{cert}(q_{i-1}, D, \Sigma)$. By Proposition 2.1, the latter boils down to showing that there exists a homomorphism λ that maps $\mathsf{atoms}(q_{i-1})$ to $\mathsf{chase}(D, \Sigma)$ and $\lambda(\bar{x}_{i-1}) = \bar{c}$, where $\bar{x}_{i-1}$ are the output variables of q_{i-1}.

Since, by hypothesis, $\bar{c} \in \mathsf{cert}(q_i, D, \Sigma)$, we conclude that there exists a homomorphism h such that $h(\mathsf{atoms}(q_i)) \subseteq \mathsf{chase}(D, \Sigma)$ and $h(\bar{x}_i) = \bar{c}$ with $\bar{x}_i$ being the output variables of q_i. Recall that q_i is a σ-resolvent of q_{i-1} for some $\sigma \in \Sigma$, i.e., q_i is such that $\mathsf{atoms}(q_i) = \gamma((\mathsf{atoms}(q_{i-1}) \setminus S) \cup \mathsf{body}(\sigma))$, while its free variables are $\gamma(\bar{x}_{i-1})$, for a MGCU (S, S', γ) of q_{i-1} with σ. Let $\mu = h \circ \gamma$. Observe that $\mu(\mathsf{body}(\sigma)) \subseteq \mathsf{chase}(D, \Sigma)$. Thus, $\mu'(\mathsf{head}(\sigma)) \subseteq \mathsf{chase}(D, \Sigma)$, where $\mu' \supseteq \mu$ maps each existentially quantified variable of σ to a fresh null. We define the substitution

$$h' \;=\; h \;\cup\; \{\gamma(z) \mapsto \mu'(z)\}_{z \in \mathsf{var}(S') \cap \mathsf{var}_\exists(\sigma)}$$

We proceed to show that

(1) h' is a well-defined substitution.
(2) The homomorphism λ such that $\lambda(\mathsf{atoms}(q_{i-1})) \subseteq \mathsf{chase}(D, \Sigma)$ and $\lambda(\bar{x}_{i-1}) = \bar{c}$ is $h' \circ \gamma$.

To show that h' is a well-defined substitution, it suffices to show that, for each $z \in \mathsf{var}(S') \cap \mathsf{var}_\exists(\sigma)$, $\gamma(z)$ is not a constant, and $\gamma(z)$ does not occur in the domain of h. By contradiction, assume that $\gamma(z)$ is either a constant, or is in the domain of h. It is easy to verify that in this case there exists $z \in \mathsf{var}(S') \cap \mathsf{var}_\exists(\sigma)$ such that $\gamma(z)$ is a constant, or $\gamma(z) = \gamma(y)$ for a variable y that is in S', or in S but shared. This contradicts the fact that (S, S', γ) is a chunk unifier.

We proceed to show that the desired homomorphism λ is $h' \circ \gamma$. Clearly, $h'(\gamma(\mathsf{atoms}(q_{i-1}) \setminus S)) \subseteq \mathsf{chase}(D, \Sigma)$. Moreover, $h'(\gamma(S)) = h'(\gamma(S')) = \mu'(S') \subseteq \mathsf{chase}(D, \Sigma)$ since $S' \subseteq \mathsf{head}(\sigma)$ and $\mu'(\mathsf{head}(\sigma)) \subseteq \mathsf{chase}(D, \Sigma)$. Finally, since $\gamma(\bar{x}_{i-1}) = \bar{x}_i$ and $h(\bar{x}_i) = \bar{c}$, we get that $h'(\gamma(\bar{x}_{i-1})) = \bar{c}$, and the claim follows. This completes the proof of Lemma A.2. □

We can now show that (2) ⇒ (1). By hypothesis, there exists a branch $(q_i)_{i\in\{0,\ldots,n\}}$, for some $n \geq 0$, of the unfolding of q with Σ such that $\bar{c} \in q_n(D)$. Therefore, $\bar{c} \in \mathsf{cert}(q_n, D, \Sigma)$ due to the monotonicity of CQs. By Lemma A.2 we get that $\bar{c} \in \mathsf{cert}(q, D, \Sigma)$, and the claim follows.

B PROOF OF LEMMA 4.15, LEMMA 4.16 AND LEMMA 4.17

Before giving the proofs of the auxiliary lemmas underlying the main Theorems 4.12 and 4.13, we first need to properly define the notion of unravelling of the chase graph, which has been only informally discussed in the main body of the paper.

B.1 Unraveling the Chase Graph

Consider a database D and a set Σ of TGDs, and let $\mathcal{G}^{D,\Sigma} = (V, E, \mu)$ be the chase graph for D and Σ; recall that the notion of chase graph has been already defined in the main body of the paper. The *unraveling of* $\mathcal{G}^{D,\Sigma}$ *around* v, with $v \in V$,[14] is the directed tree $\mathcal{G}^{D,\Sigma}_v = (V_v, E_v)$, where

[14]Recall that the nodes of $\mathcal{G}^{D,\Sigma}$ are actually atoms from $\mathsf{chase}(D, \Sigma)$.

- V_v is the set of all finite sequences $\bar{v} = v_1 v_2 \cdots v_n$ of nodes from V such that $v_1 = v$ and $(v_{i+1}, v_i) \in E$ (we may also write $v_{i+1} E v_i$) for all $i \in [n-1]$; we write $\text{last}(\bar{v})$ for v_n.
- For $\bar{v} = v_1 \cdots v_n$ and $\bar{w} = v_1 \cdots v_n v_{n+1}$ we have that $\bar{v} E_v \bar{w}$ iff $v_{n+1} E v_n$.

Given a set $\Theta \subseteq V$ of nodes, the *unraveling of* $\mathcal{G}^{D,\Sigma}$ *around* Θ is the directed node- and edge-labeled forest $\mathcal{G}^{D,\Sigma}_\Theta = (V_\Theta, E_\Theta, \mu_\Theta)$, where $V_\Theta = \bigcup_{v \in \Theta} V_v$ and $E_\Theta = \bigcup_{v \in \Theta} E_v$. Before giving the definition of the labeling function μ_Θ, we need to introduce some auxiliary notions.

A *pseudo path* in $\mathcal{G}^{D,\Sigma}_\Theta$ is a sequence of nodes $\bar{v}_1, \ldots, \bar{v}_n$ from V_Θ such that, for each $i \in [n-1]$,

$$\bar{v}_i E_\Theta \bar{v}_{i+1} \quad \text{or} \quad \bar{v}_{i+1} E_\Theta \bar{v}_i \quad \text{or} \quad |\bar{v}_i| = |\bar{v}_{i+1}| = 1.$$

In other words, a pseudo path in $\mathcal{G}^{D,\Sigma}_\Theta$ is a path in $\mathcal{G}^{D,\Sigma}_\Theta$ where we consider the root nodes of the forest to be connected. Note that there is a unique shortest pseudo path between any two nodes of $\mathcal{G}^{D,\Sigma}_\Theta$. Let $\bar{v}$ and $\bar{w}$ be nodes from $\mathcal{G}^{D,\Sigma}_\Theta$. If $\bar{v}$ and $\bar{w}$ lie in the same tree component $\mathcal{T}$, then we denote by $\text{gca}(\bar{v}, \bar{w})$ the singleton set consisting of the greatest common ancestor of $\bar{v}$ and $\bar{w}$ in $\mathcal{T}$. If, on the other hand, $\bar{v}$ and $\bar{w}$ are in different tree components, then $\text{gca}(\bar{v}, \bar{w}) = \emptyset$. Given a term t, we say that $\bar{v}$ and $\bar{w}$ are *t-connected in* $\mathcal{G}^{D,\Sigma}_\Theta$ if one of the following holds:

(1) t is a constant, and t occurs in $\text{last}(\bar{v})$ and in $\text{last}(\bar{w})$.
(2) t occurs in $\text{last}(\bar{u})$ for every $\bar{u} \notin \text{gca}(\bar{v}, \bar{w})$ that lies on the unique shortest pseudo path between $\bar{v}$ and $\bar{w}$ in $\mathcal{G}^{D,\Sigma}_\Theta$.

Clearly, this relation defines an equivalence relation among the nodes of $\mathcal{G}^{D,\Sigma}_\Theta$. We write $[\bar{v}]_t$ for the respective equivalence class of $\bar{v} \in V_\Theta$.[15] Moreover, if a is a constant, then, since $[\bar{v}]_a = [\bar{w}]_a$ for any $\bar{v}, \bar{w} \in V_\Theta$, we identify the class $[\bar{v}]_a$ simply with a. For $[\bar{v}]_t$ with t being a labeled null, we call $[\bar{v}]_t$ a labeled null as well. We are now ready to define the labeling function μ_Θ. Consider a node $\bar{v} \in V_\Theta$, and assume that $\text{last}(\bar{v}) = R(t_1, \ldots, t_k)$. Then, we define

$$\mu_\Theta(\bar{v}) \;=\; R([\bar{v}]_{t_1}, \ldots, [\bar{v}]_{t_k}).$$

Moreover, if $\bar{v} E_\Theta \bar{w}$ and $\mu(\text{last}(\bar{v}), \text{last}(\bar{w})) = (\sigma, h)$, we define

$$\mu_\Theta(\bar{v}, \bar{w}) \;=\; (\sigma, h^*),$$

where, for a variable x in $\text{body}(\sigma)$, $h^*(x) = [\bar{v}]_{h(x)}$.

We write $U(\mathcal{G}^{D,\Sigma}, \Theta)$ for the instance $\bigcup_{\bar{v} \in V_\Theta} \mu_\Theta(\bar{v})$. Notice that, since we identify $[\bar{v}]_a$ with a, when a is a constant this means that $R(a_1, \ldots, a_n) \in U(\mathcal{G}^{D,\Sigma}, \Theta)$ for every fact $R(a_1, \ldots, a_n) \in \mathcal{G}^{D,\Sigma} \restriction \Theta$ with the latter being the set of all atoms that lie on some path in $\mathcal{G}^{D,\Sigma}$ that leads from a database atom to some atom of Θ. Given a node $\bar{v}$ of $\mathcal{G}^{D,\Sigma}_\Theta$, we denote by $\text{succ}_{\sigma,h}(\bar{v})$ the set of labels of all children of $\bar{v}$ whose edge from $\bar{v}$ is labeled by (σ, h). Accordingly, we write $\text{succ}(\bar{v})$ for the set of all labels of children of $\bar{v}$. When using this notation, we assume that the particular unraveling we are referring to is clear from context. We write $\alpha_1, \ldots, \alpha_k \Rightarrow_{\sigma,h} \beta$ if there is a node $\bar{v}$ of $\mathcal{G}^{D,\Sigma}_\Theta$ such that $\mu_\Theta(\bar{v}) = \beta$ and $\text{succ}_{\sigma,h}(\bar{v}) = \{\alpha_1, \ldots, \alpha_k\}$. Accordingly, we write $\alpha_1, \ldots, \alpha_k \Rightarrow \beta$ if there is a $\sigma \in \Sigma$ and some h such that $\alpha_1, \ldots, \alpha_k \Rightarrow_{\sigma,h} \beta$. The next technical lemma follows by the definitions of the equivalence classes $[v]_t$ and the labeling function μ_Θ.

Lemma B.1. *Consider a node $\bar{v}$ of $\mathcal{G}^{D,\Sigma}_\Theta$ with $\text{succ}_{\sigma,h}(\bar{v}) = \{\beta_1, \ldots, \beta_k\}$ for some $\sigma \in \Sigma$ and h. Then, if some labeled null occurs in β_i, it either occurs also in $\mu_\Theta(\bar{v})$, or it does not occur in the label of any node of $\mathcal{G}^{D,\Sigma}_\Theta$ that is not a descendant of $\bar{v}$.*

Note that there is an obvious homomorphism h_Θ from Θ to $U(\mathcal{G}^{D,\Sigma}, \Theta)$: for each $t \in \text{dom}(\Theta)$, $h_\Theta(t) = [\bar{v}_0]_t$, where $\bar{v}_0$ is any of the root nodes. It is clear that h_Θ is well-defined since $[\bar{v}_0]_t = [\bar{w}_0]_t$

[15] Formally, we define $[\bar{v}]_t = \{(\bar{u}, t) \mid \bar{u} \text{ is } t\text{-connected to } \bar{v}\}$ to ensure that $[\bar{v}]_t = [\bar{w}]_{t'}$ only if $t = t'$.

for all root nodes $\bar{v}_0, \bar{w}_0$ of $\mathcal{G}^{D,\Sigma}_\Theta$, and all terms $t \in \mathsf{dom}(\Theta)$. The mapping h_Θ is indeed a homomorphism since, for $\alpha = R(t_1, \ldots, t_n) \in \Theta$, $h_\Theta(R(t_1, \ldots, t_n)) = R([\alpha]_{t_1}, \ldots, [\alpha]_{t_n}) \in U(\mathcal{G}^{D,\Sigma}, \Theta)$.

Blocking, Depth and Rank. Consider an atom $\alpha \in U(\mathcal{G}^{D,\Sigma}, \Theta)$, and suppose that $\beta_1, \ldots, \beta_k \Rightarrow \alpha$. For a set $\Gamma \subseteq U(\mathcal{G}^{D,\Sigma}, \Theta)$, we say that the *application of* $\beta_1, \ldots, \beta_k \Rightarrow \alpha$ *is blocked in* Γ if there is a labeled null occurring in α that occurs in $\Gamma \setminus \{\alpha\}$, but that does not occur in any of $\beta_1, \ldots, \beta_k$. Given a node $\bar{v}$ of $\mathcal{G}^{D,\Sigma}_\Theta$, the *depth* of $\bar{v}$, denoted $\mathsf{dp}(\bar{v})$, is defined inductively as follows:

$$\mathsf{dp}(\bar{v}) \;=\; \max\left\{\mathsf{dp}(\bar{u}) \mid \bar{u} \text{ is a child node of } \bar{v} \text{ in } \mathcal{G}^{D,\Sigma}_\Theta\right\} + 1.$$

For an atom $\alpha \in U(\mathcal{G}^{D,\Sigma}, \Theta)$, we define the *depth* of α as

$$\mathsf{dp}(\alpha) \;=\; \min\{\mathsf{dp}(\bar{v}) \mid \mu_\Theta(\bar{v}) = \alpha\}\,.$$

Observe that $\mathsf{dp}(\alpha) = 1$ iff α labels only leaf nodes in $\mathcal{G}^{D,\Sigma}_\Theta$. For a set of atoms $\Gamma \subseteq U(\mathcal{G}^{D,\Sigma}, \Theta)$, let $\mathsf{dp}(\Gamma) = \max\{\mathsf{dp}(\alpha) \mid \alpha \in \Gamma\}$. Finally, the *rank* of $\bar{v}$, denoted $\mathsf{rk}(\bar{v})$, is defined as

$$\mathsf{rk}(\bar{v}) \;=\; \begin{cases} 1 & \text{if } \bar{v} \text{ is a leaf node in } \mathcal{G}^{D,\Sigma}_\Theta, \\ \sum_{\bar{u} \in \mathsf{succ}(\bar{v})} \mathsf{rk}(\bar{u}) & \text{otherwise.} \end{cases}$$

For an atom $\alpha \in U(\mathcal{G}^{D,\Sigma}, \Theta)$, we define the *rank* of α as

$$\mathsf{rk}(\alpha) \;=\; \min\{\mathsf{rk}(v) \mid \mu_\Theta(v) = \alpha\}\,.$$

For a set of atoms $\Gamma \subseteq U(\mathcal{G}^{D,\Sigma}, \Theta)$, let $\mathsf{rk}(\Gamma) = \sum_{\alpha \in \Gamma} \mathsf{rk}(\alpha)$. Intuitively, the rank of an atom measures how many database facts are used to derive that atom.

Both the depth and rank will be used as induction parameters in the subsequent proofs. We remark that those parameters are, of course, always defined relative to a particular unraveling of $\mathcal{G}^{D,\Sigma}$. The concrete unraveling U of $\mathcal{G}^{D,\Sigma}$ that they refer to will always be clear from context, and thus, we do not need an additional notation to make the connection with U explicit.

B.2 Proof of Lemma 4.15

Let us first recall the statement:

Consider a database D, and a set $\Sigma \in \mathsf{WARD} \cap \mathsf{PWL}$ of single-head TGDs in level-wise normal form. Let $\Theta \subseteq \mathsf{chase}(D, \Sigma)$ and $\Gamma \subseteq U(\mathcal{G}^{D,\Sigma}, \Theta)$. There exists a linear chase tree $\mathcal{C}$ for Γ such that $\mathsf{nwd}(\mathcal{C}) \leq f_{\mathsf{WARD}\cap\mathsf{PWL}}(\Gamma, \Sigma)$.

Recall that for a predicate $P \in \mathsf{sch}(\Sigma)$, we write $\ell_\Sigma(P)$ for the level of P w.r.t. Σ. We also write $\ell_\Sigma(\alpha)$ for the level of the predicate of α. In what follows, for brevity, we omit the subindex Σ, i.e., we simply write $\ell(\cdot)$ instead of $\ell_\Sigma(\cdot)$. We also write $\ell(\Sigma)$ for $\max\{\ell_\Sigma(P) \mid P \in \mathsf{sch}(\Sigma)\}$, and we write b for $\max\{|\mathsf{body}(\sigma)| \mid \sigma \in \Sigma\}$. Before we proceed with the proof of Lemma 4.15, let us introduce some additional technical notions that will be useful for the proof.

Conflict-free Atoms. The *stratification of* Σ is a partition $\{\mathcal{S}_1, \ldots, \mathcal{S}_{\ell(\Sigma)}\}$ of $\mathsf{sch}(\Sigma)$ such that, for each $P \in \mathsf{sch}(\Sigma)$, we have that $P \in \mathcal{S}_k$ iff $\ell_\Sigma(P) = k$. Let $\{\Gamma[\mathcal{S}_1], \ldots, \Gamma[\mathcal{S}_{\ell(\Sigma)}]\}$be the unique partition of Γ such that, for every $\alpha \in \Gamma$, we have that $\alpha \in \Gamma[\mathcal{S}_k]$ iff $\ell(\alpha) = k$. The *join graph* of Γ is the undirected edge-labeled graph $\mathbb{J}(\Gamma)$ whose set of nodes is Γ, and that has an edge between two distinc atoms α and β labeled with terms $t_1, \ldots, t_k$ iff $t_1, \ldots, t_k$ are all the terms occurring in both α and β. We say that an atom $\alpha \in \Gamma[\mathcal{S}_i]$ is *conflict-free (in Γ)* if there is no atom $\beta \in \bigcup_{j>i} \Gamma[\mathcal{S}_j]$ such that there is a path from α to β in $\mathbb{J}(\Gamma)$ that has an occurrence of a null in each of its edge labels – we shall call such a path *conflicting*. Hence, an atom that only has constants as arguments

is trivially conflict-free. Notice also that Γ trivially has conflict-free atoms, namely those that are among $\Gamma[\mathcal{S}_r]$, where r is the largest number such that $\Gamma[\mathcal{S}_r]$ is non-empty.

Size Measures. We define the value

$$l_\Gamma \;=\; \min\{k \mid k \in [\ell(\Sigma)] \text{ and } \Gamma[\mathcal{S}_k] \text{ contains an atom that is conflict-free}\}\,.$$

Let

$$a_1^\Gamma, \ldots, a_{\ell(\Sigma)}^\Gamma \quad \text{and} \quad n_1^\Gamma, \ldots, n_{\ell(\Sigma)}^\Gamma$$

be sequences of integers such that $a_i^\Gamma = |\Gamma[\mathcal{S}_i]|$ and n_i^Γ is the number of atoms from $\Gamma[\mathcal{S}_i]$ that are *not* conflict-free. We finally define the value

$$m_\Gamma \;=\; \sum_{i=1}^{\ell(\Sigma)} b \cdot \max\left\{a_i^\Gamma, 1 + n_i^\Gamma\right\}.$$

We are now ready to give the proof of Lemma 4.15.

The Proof. Our goal is to show that there exists a linear chase tree C for Γ (w.r.t. $\mathcal{G}_\Theta^{D,\Sigma}$) whose node-width is bounded by m_Γ. This suffices to prove the claim since $m_\Gamma \leq f_{\mathsf{WARD}\cap\mathsf{PWL}}(\Gamma, \Sigma)$. The proof is by induction on the rank of Γ, i.e., the value $\mathsf{rk}(\Gamma)$.

Base Case. Suppose first that $\mathsf{rk}(\Gamma) = 1$. The *level depth* of Γ is defined as

$$\mathsf{ldp}(\Gamma) \;=\; \max\left\{\mathsf{dp}(\alpha) \mid \alpha \in \Gamma[\mathcal{S}_{l_\Gamma}] \text{ and } \alpha \text{ is conflict-free}\right\}.$$

We perform an auxiliary induction on $\mathsf{ldp}(\Gamma)$ in order to prove our claim:

- Suppose first that $\mathsf{ldp}(\Gamma) = 1$. Since $\mathsf{rk}(\Gamma) = 1$, this means that actually $\Gamma = \{\alpha\}$ for some fact $\alpha \in D$. A linear chase tree for Γ of node-width at most $m_\Gamma \geq |\Gamma|$ is simply the tree with a single root node whose label is Γ.
- Suppose now that $\mathsf{rk}(\Gamma) = 1$ and $\mathsf{ldp}(\Gamma) = n + 1$. This means that $\Gamma = \{\alpha\}$ for some atom α such that $\beta \Rightarrow \alpha$ for some atom β with $\mathsf{dp}(\beta) = n$. Note that the application of $\beta \Rightarrow \alpha$ is trivially not blocked. We let $\Gamma' = \{\beta\}$, and we observe that $\mathsf{ldp}(\Gamma) = n$. By induction hypothesis, there is a linear chase tree C' for Γ' whose node-width is bounded by $m_{\Gamma'}$. Let C be the linear chase tree whose root v_0 is labeled with Γ such that v_0 has a single child labeled with Γ'. It is easy to check that $m_\Gamma = m_{\Gamma'} = b$. Therefore, C is a linear chase tree for Γ whose node-width is bounded by m_Γ, as needed.

Induction Step. Suppose now that $\mathsf{rk}(\Gamma) = m + 1$. A *reduction sequence for* Γ is a finite sequence $\Gamma_0, \Gamma_1, \ldots, \Gamma_k, \Xi$ of sets of atoms that satisfy the following conditions:

(1) $\Gamma_0 = \Gamma$,
(2) $m_{\Gamma_0} \geq m_{\Gamma_1} \geq \cdots \geq m_{\Gamma_k}$,
(3) each Γ_{i+1} is an unfolding of Γ_i, where $i \in \{0, \ldots, k-2\}$,
(4) $\mathsf{rk}(\Gamma_k) \leq m$, and
(5) $\Xi = \Gamma_{k-1} \cap D$ and $\Gamma_k = \Gamma_{k-1} \setminus \Xi$; thus, $\{\Gamma_k, \Xi\}$ is a decomposition of Γ_{k-1}.

We proceed to show that there is a reduction sequence for Γ, which in turn will allow us to easily complete the induction step, and thus, the proof of Lemma 4.15.

Lemma B.2. *There exists a reduction sequence for* Γ.

Proof. We shall also perform a subsidiary induction on $\mathsf{ldp}(\Gamma)$ to show that, for every $n \geq 1$, if $\mathsf{ldp}(\Gamma) = n$, then there exists a reduction sequence for Γ.

Suppose first that $\mathsf{ldp}(\Gamma) = 1$

In this case, Γ contains at least one fact from D. Let $\{\alpha_1, \ldots, \alpha_k\} = \Gamma \cap D$. We form the reduction sequence $\pi = \Gamma, \Gamma', \{\alpha_1, \ldots, \alpha_k\}$, where $\Gamma' = \Gamma \setminus \{\alpha_1, \ldots, \alpha_k\}$. Notice that $\mathrm{rk}(\Gamma') \leq m$. In order to prove that π is indeed a reduction sequence for Γ, it remains to show that $m_{\Gamma'} \leq m_\Gamma$. Observe that

$$\begin{aligned} m_\Gamma - m_{\Gamma'} &= \sum_{i=1}^{\ell(\Sigma)} b \cdot \max\left\{a_i^\Gamma, 1 + n_i^\Gamma\right\} - \sum_{i=1}^{\ell(\Sigma)} b \cdot \max\left\{a_i^{\Gamma'}, 1 + n_i^{\Gamma'}\right\} \\ &= \underbrace{b \cdot \max\left\{a_{l_\Gamma}^\Gamma, 1 + n_{l_\Gamma}^\Gamma\right\}}_{= a_{l_\Gamma}^\Gamma \text{ since } k \geq 1} - b \cdot \max\left\{ \underbrace{a_{l_\Gamma}^{\Gamma'}}_{= a_{l_\Gamma}^\Gamma - k}, \overbrace{1 + n_{l_\Gamma}^{\Gamma'}}^{= 1 + n_{l_\Gamma}^\Gamma} \right\}. \end{aligned}$$

If $a_{l_\Gamma}^\Gamma - k \geq 1 + n_{l_\Gamma}^\Gamma$, then we obtain that

$$m_\Gamma - m_{\Gamma'} = b a_{l_\Gamma}^\Gamma - b(a_{l_\Gamma}^\Gamma - k) = bk > 0,$$

and if $a_{l_\Gamma}^\Gamma - k < 1 + n_{l_\Gamma}^\Gamma$, then we obtain that

$$m_\Gamma - m_{\Gamma'} = b a_{l_\Gamma}^\Gamma - b(1 + n_{l_\Gamma}^\Gamma) \geq 0, \quad \text{since } 1 + n_{l_\Gamma}^\Gamma \leq a_{l_\Gamma}^\Gamma.$$

Therefore, in both cases we have that $m_{\Gamma'} \leq m_\Gamma$, as needed.

Suppose now that $\mathrm{ldp}(\Gamma) = n + 1$

Assume that $\alpha_1, \ldots, \alpha_k$ enumerates the conflict-free atoms from $\Gamma[\mathcal{S}_{l_\Gamma}]$ of maximal depth (i.e., $n+1$). For $i \in [k]$, let $\beta_{i,1}, \ldots, \beta_{i,k_i}$ be atoms such that the application of $\beta_{i,1}, \ldots, \beta_{i,k_i} \Rightarrow \alpha_i$ is not blocked in Γ; those atoms exist since each α_i is conflict-free in Γ and of maximal depth. Let σ_i and h_i be such that $\beta_{i,1}, \ldots, \beta_{i,k_i} \Rightarrow_{\sigma_i, h_i} \alpha_i$. Let $\Gamma_0 = \Gamma$ and $\Gamma_i = (\Gamma_{i-1} \setminus \{\alpha_i\}) \cup \{\beta_{i,1}, \ldots, \beta_{i,k_i}\}$, for each $i \in [k]$.

Claim 1. *For each $i \in [k]$, either $l_{\Gamma_i} = l_{\Gamma_{i-1}}$ or $l_{\Gamma_i} = l_{\Gamma_{i-1}} - 1$.*

Proof. Let us first remark that, since Σ is in level-wise normal form, we get that $\ell(\beta_{i,j}) \in \{\ell(\alpha_i), \ell(\alpha_i) - 1\}$ for each $j \in [k_i]$. We proceed by case analysis.

Suppose first that α_i has no labeled nulls as arguments. In this case, all the nulls occurring in $\beta_{i,1}, \ldots, \beta_{i,k_i}$ must be new in the sense that they do not appear in Γ_{i-1} (see Lemma B.1). Therefore, at least one of $\beta_{i,1}, \ldots, \beta_{i,k_i}$ must be conflict-free in Γ_i, and hence $l_{\Gamma_i} = l_{\Gamma_{i-1}}$ or $l_{\Gamma_i} = l_{\Gamma_{i-1}} - 1$, since Σ is in level-wise normal form, as needed.

Suppose now that α_i has labeled nulls as arguments. Since Σ is warded, σ_i has at most one atom in its body (the ward) that shares nulls with α_i. Assuming that σ_i has no ward in its body, then all the nulls that appear in α_i are not present in $\beta_{i,1}, \ldots, \beta_{i,k_i}$ (Lemma B.1) and, moreover, all the nulls that appear in $\beta_{i,1}, \ldots, \beta_{i,k_i}$ do not occur in Γ_{i-1}. Hence, at least one of $\beta_{i,1}, \ldots, \beta_{i,k_i}$ must be conflict-free in Γ_i, which implies that $l_{\Gamma_i} = l_{\Gamma_{i-1}}$ or $l_{\Gamma_i} = l_{\Gamma_{i-1}} - 1$. Suppose now that $\beta_{i,j}$ is the ward among $\beta_{i,1}, \ldots, \beta_{i,k_i}$. If $k_i \geq 2$, then the claim is immediate since the atoms from $\{\beta_{i,1}, \ldots, \beta_{i,k_i}\} \setminus \{\beta_{i,j}\}$ are all conflict-free in Γ_i due to the fact that their nulls do not appear in Γ_{i-1} by Lemma B.1. Suppose now that $k_i = 1$. Then, $\beta_{i,j}$ is conflict-free in Γ_i or not. In the former case, we immediately obtain $l_{\Gamma_i} \leq l_{\Gamma_{i-1}}$, and thus $l_{\Gamma_i} = l_{\Gamma_{i-1}}$ or $l_{\Gamma_i} = l_{\Gamma_{i-1}} - 1$ since Σ is in level-wise normal form. In the latter case, $\beta_{i,j}$ is connected to some atom $\gamma \in \bigcup_{r > \ell(\beta_{i,j})} \Gamma_i[\mathcal{S}_r]$ in the join graph of Γ_{i-1} via a path that is conflicting. Notice again that all the nulls that occur in $\beta_{i,j}$ are either only contained in $\beta_{i,j}$ or they also appear in α_i (see Lemma B.1). Hence, there is also a conflicting path in $\mathbb{J}(\Gamma_{i-1})$ that connects α_i and γ. Since α_i was assumed to be conflict-free in Γ_{i-1}, it follows that $\gamma \in \Gamma_{i-1}[\mathcal{S}_1] \cup \cdots \cup \Gamma_{i-1}[\mathcal{S}_{\ell(\alpha)}]$.

Moreover, this entails that γ is conflict-free in Γ_{i-1}. Now, γ must also be conflict-free in Γ_i since all the nulls present in Γ_i that do not occur in Γ_{i-1} must be solely contained in $\beta_{i,j}$. Thus, $l_{\Gamma_i} \leq l_{\Gamma_{i-1}}$, and hence $l_{\Gamma_i} = l_{\Gamma_{i-1}}$ or $l_{\Gamma_i} = l_{\Gamma_{i-1}} - 1$ since Σ is in level-wise normal form.

This completes the proof of Claim 1. □

The next three technical claims establish some useful properties when $l_{\Gamma_i} = l_{\Gamma_{i-1}}$ or $l_{\Gamma_i} = l_{\Gamma_{i-1}} - 1$, for $i \in [k]$ (which we know from Claim 1 that are the only possible cases), which in turn will allow us to establish Claim 5 that is crucial for completing the proof of Lemma B.2.

Claim 2. *Assume that $l_{\Gamma_i} = l_{\Gamma_{i-1}} - 1$, and that there is an atom $\beta_{i,j}$ that is not conflict-free in Γ_i. Then, there is an atom $\gamma \in \Gamma_i[\mathcal{S}_{l_{\Gamma_{i-1}}}]$ that is conflict free in Γ_i.*

Proof. Observe that $\ell(\beta_{i,j}) = l_{\Gamma_i}$ since all the nulls in $\beta_{i,j}$ either are also in α_i, or are fresh. Hence, if we had $\ell(\beta_{i,j}) = l_{\Gamma_{i-1}} = l_{\Gamma_i} + 1$, then α would not be conflict-free as well. Now, since $\beta_{i,j}$ is not conflict-free, it follows that $\beta_{i,j}$ is connected via a conflicting path to some $\gamma \in \bigcup_{r > \ell(\beta_{i,j})} \Gamma_i[\mathcal{S}_r]$. It is easy to see that, in fact, we must have $\ell(\gamma) = \ell(\beta_{i,j}) + 1 = l_{\Gamma_{i-1}}$. Therefore, $\beta_{i,j}$ shares a null with γ, and thus γ shares a null with α_i. Hence, γ must be conflict-free since α_i is. □

Claim 3. *Assume that $l_{\Gamma_i} = l_{\Gamma_{i-1}} = 1$. Then, $k_i = 1$.*

Proof. Since $\mathsf{ldp}(\alpha_i) = n + 1 \geq 2$, $\ell(\alpha_i) = 1$ only if α_i is derived by a sequence of atoms that have the same predicate as α_i. Hence, necessarily $k_i = 1$ since Σ is piece-wise linear. □

Claim 4. *Assume that $l = l_{\Gamma_i} = l_{\Gamma_{i-1}} > 1$. Then, $|\Gamma_i[\mathcal{S}_l] \setminus \Gamma_{i-1}[\mathcal{S}_{l-1}]| \leq 1$, i.e., $a^{\Gamma_i}_{l-1} - a^{\Gamma_{i-1}}_{l-1} \leq 1$.*

Proof. Towards a contradiction, assume that $a^{\Gamma_i}_{l-1} - a^{\Gamma_{i-1}}_{l-1} > 1$, i.e., at least two of the $\beta_{i,1}, \ldots, \beta_{i,k_i}$ have level $l - 1$. Since $l_{\Gamma_i} = l_{\Gamma_{i-1}}$, this means that they are actually not conflict-free, and thus they share a null with α_i. But this contradicts the fact that Σ is warded, which in turn implies that at most one of the atoms $\beta_{i,1}, \ldots, \beta_{i,k_i}$ can share nulls with the atom α_i. □

We are now ready, by exploiting Claims 2, 3 and 4, to show the following key claim.

Claim 5. *For each $i \in [k]$, it holds that $m_{\Gamma_i} \leq m_{\Gamma_{i-1}}$.*

Proof. For the sake of readability, let $l = l_{\Gamma_{i-1}}$, $l' = l_{\Gamma_i}$, $\Gamma = \Gamma_{i-1}$ and $\Gamma' = \Gamma_i$. We proceed by considering the cases $l' = l$ and $l' = l - 1$, which, by Claim 1, are the only valid cases.

Case 1: Suppose that $l' = l$. We proceed by considering the following subcases:

Subcase 1.1: Suppose first that $l = l' = 1$. Then,

$$\begin{aligned} m_\Gamma - m_{\Gamma'} &= b \cdot \max\left\{a^\Gamma_1, 1 + n^\Gamma_1\right\} - b \cdot \max\left\{a^{\Gamma'}_1, 1 + n^{\Gamma'}_1\right\} \\ &= b a^\Gamma_1 - b a^{\Gamma'}_1 \\ &= b a^\Gamma_1 - b(a^\Gamma_1 + k_i - 1) \\ &= 0 \quad \text{(since } k_i = 1 \text{ by Claim 3).} \end{aligned}$$

Hence, $m_{\Gamma'} = m_\Gamma$.

Subcase 1.2: Suppose that $l > 1$. Then, one can verify that

$$\begin{aligned} m_\Gamma - m_{\Gamma'} = {} & b \cdot \max\left\{a^\Gamma_l, 1 + n^\Gamma_l\right\} + b \cdot \max\left\{a^\Gamma_{l-1}, 1 + n^\Gamma_{l-1}\right\} \\ & - b \cdot \max\left\{a^{\Gamma'}_l, 1 + n^{\Gamma'}_l\right\} - b \cdot \max\left\{a^{\Gamma'}_{l-1}, 1 + n^{\Gamma'}_{l-1}\right\}. \end{aligned}$$

By Claim 4, we know that $a_{l-1}^{\Gamma'} \leq a_{l-1}^{\Gamma} + 1$. In case $a_{l-1}^{\Gamma'} = a_{l-1}^{\Gamma}$, we get that $m_\Gamma - m_{\Gamma'} \geq 0$ by observing that $a_l^{\Gamma} = a_l^{\Gamma'}$. Assume now that $a_{l-1}^{\Gamma'} = a_{l-1}^{\Gamma} + 1$, and also observe that we must have $n_{l-1}^{\Gamma'} = a_{l-1}^{\Gamma'}$ and $a_{l-1}^{\Gamma} = n_{l-1}^{\Gamma}$ since, by assumption, Γ and Γ' do not have any conflict-free atoms of level $l-1$. Moreover, notice that $n_l^{\Gamma'} = n_l^{\Gamma}$. From the above, we obtain that

$$\begin{aligned} m_\Gamma - m_{\Gamma'} &= b \cdot \max\left\{a_l^{\Gamma'} + k_i, 1 + n_l^{\Gamma'}\right\} + b \cdot \max\left\{a_{l-1}^{\Gamma'} - 1, a_{l-1}^{\Gamma'}\right\} \\ &\quad - \underbrace{b \cdot \max\left\{a_l^{\Gamma'}, 1 + n_l^{\Gamma'}\right\}}_{= a_l^{\Gamma'} \text{ by Claim 2}} - b \cdot \max\left\{a_{l-1}^{\Gamma'}, 1 + a_{l-1}^{\Gamma'}\right\} \\ &= b(a_l^{\Gamma'} + k_i) + b \cdot a_{l-1}^{\Gamma'} - b(a_{l-1}^{\Gamma'} - 1) - b \cdot a_l^{\Gamma'} \\ &= b(k_i - 1) \\ &\geq 0. \end{aligned}$$

This proves the claim for the case $l = l' > 1$.

Case 2: Suppose that $l' = l - 1$. Observe that

$$\begin{aligned} m_\Gamma - m_{\Gamma'} &= \sum_{i=1}^{\ell(\Sigma)} b \cdot \max\left\{a_i^{\Gamma}, 1 + n_i^{\Gamma}\right\} - \sum_{i=1}^{\ell(\Sigma)} b \cdot \max\left\{a_i^{\Gamma'}, 1 + n_i^{\Gamma'}\right\} \\ &= b \cdot \max\left\{a_{l'}^{\Gamma}, 1 + n_{l'}^{\Gamma}\right\} - b \cdot \max\left\{a_{l'}^{\Gamma'}, 1 + n_{l'}^{\Gamma'}\right\} \\ &\quad + \underbrace{b \cdot \max\left\{a_l^{\Gamma}, 1 + n_l^{\Gamma}\right\}}_{= a_l^{\Gamma} \text{ by assumption}} - b \cdot \max\left\{a_l^{\Gamma'}, 1 + n_l^{\Gamma'}\right\} \\ &\geq b \cdot \max\left\{a_{l'}^{\Gamma}, 1 + n_{l'}^{\Gamma}\right\} - b \cdot \max\{a_{l'}^{\Gamma'}, 1 + n_{l'}^{\Gamma'}\}, \end{aligned}$$

where the last inequality holds since $a_l^{\Gamma'} \leq a_l^{\Gamma}$ by piece-wise linearity, and thus

$$\max\left\{a_l^{\Gamma'}, 1 + n_l^{\Gamma'}\right\} = \max\left\{a_l^{\Gamma'}, 1 + n_l^{\Gamma}\right\} \leq a_l^{\Gamma}.$$

By construction of Γ', we know that $a_{l'}^{\Gamma} = n_{l'}^{\Gamma}$ and $a_{l'}^{\Gamma'} \leq a_{l'}^{\Gamma} + k_i \leq a_{l'}^{\Gamma} + b$. We proceed by considering the following subcases:

Subcase 2.1: Suppose that $\max\{a_{l'}^{\Gamma'}, 1 + n_{l'}^{\Gamma'}\} = a_{l'}^{\Gamma'}$. Then,

$$m_\Gamma - m_{\Gamma'} = b(1 + n_{l'}^{\Gamma}) - b \cdot a_{l'}^{\Gamma'} \geq 0$$

since $a_{l'}^{\Gamma'} \leq a_{l'}^{\Gamma} + b$ and $n_{l'}^{\Gamma} = a_{l'}^{\Gamma}$.

Subcase 2.2: Suppose that $\max\left\{a_{l'}^{\Gamma'}, 1 + n_{l'}^{\Gamma'}\right\} = 1 + n_{l'}^{\Gamma'}$. The fact that $1 + n_{l'}^{\Gamma'} \geq a_{l'}^{\Gamma'}$ entails that the number of conflict-free atoms in $\Gamma'[\mathcal{S}_{l'}]$ is at most 1. Since $l' = l - 1$, it must actually be the case that the number of conflict-free atoms in $\Gamma'[\mathcal{S}_{l'}]$ is exactly one, i.e., $a_{l'}^{\Gamma'} - n_{l'}^{\Gamma'} = 1$. Therefore, we must have $n_{l'}^{\Gamma'} = n_{l'}^{\Gamma}$. Hence, we obtain that

$$m_\Gamma - m_{\Gamma'} = b(1 + n_{l'}^{\Gamma}) - b(1 + n_{l'}^{\Gamma'}) = 0,$$

which proves that $m_\Gamma = m_{\Gamma'}$.

This completes the proof of Claim 5. □

Let us now complete the proof of Lemma B.2. By construction, $\mathsf{ldp}(\Gamma_k) \leq n$. Hence, by induction hypothesis, there is a reduction sequence $\pi' = \Gamma_{0,k}, \Gamma_{1,k}, \ldots, \Gamma_{k',k}, \Xi$ for Γ_k such that $\Gamma_{0,k} = \Gamma_k$, $\mathsf{rk}(\Gamma_{k',k}) \leq m$, and $m_{\Gamma_{0,k}} \geq m_{\Gamma_{1,k}} \geq \cdots \geq m_{\Gamma_{k',k}}$. We define the sequence

$$\pi \;=\; \Gamma_0, \Gamma_1, \ldots, \Gamma_k, \Gamma_{1,k}, \ldots, \Gamma_{k',k}, \Xi,$$

and recall that $\Gamma_0 = \Gamma$. Claim 5 yields $m_\Gamma = m_{\Gamma_0} \geq m_{\Gamma_1} \geq \cdots \geq m_{\Gamma_k}$. Hence, it follows that

$$m_\Gamma \;=\; m_{\Gamma_0} \;\geq\; m_{\Gamma_1} \;\geq\; \cdots \;\geq\; m_{\Gamma_k} \;\geq\; m_{\Gamma_{1,k}} \;\geq\; \cdots \;\geq\; m_{\Gamma_{k',k}}.$$

Therefore, π is a reduction sequence for Γ. This concludes the induction step of our subsidiary induction on $\mathsf{ldp}(\Gamma)$, and thus the proof of Lemma B.2. □

We can now easily conclude the induction step for the induction on $\mathsf{rk}(\Gamma)$ as follows. We know by Lemma B.2 that there is a reduction sequence $\Gamma_0, \Gamma_1, \ldots, \Gamma_k, \Xi$ for Γ such that

(1) $\Gamma_0 = \Gamma$,
(2) $m_{\Gamma_0} \geq m_{\Gamma_1} \geq \cdots \geq m_{\Gamma_k}$,
(3) each Γ_{i+1} is an unfolding of Γ_i, where $i \in \{0, \ldots, k-2\}$,
(4) $\mathsf{rk}(\Gamma_k) \leq m$, and
(5) $\Xi = \Gamma_{k-1} \cap D$ and $\Gamma_k = \Gamma_{k-1} \setminus \Xi$.

Since $\mathsf{rk}(\Gamma_k) \leq m$, by induction hypothesis, there exists a linear chase tree C' for Γ_k whose node-width is bounded by m_{Γ_k}. Let C be the linear chase tree for Γ constructed as follows. The root v_0 of C is labeled Γ, and there are nodes $v_1, \ldots, v_k, v'_k$ such that (i) for each $i \in \{0, \ldots, k-1\}$, v_i is labeled with Γ_i, (ii) v_{i+1} is the only child of v_i, for each $i \in \{0, \ldots, k-2\}$, while (iii) v_{k-1} has two children, namely v_k and v'_k, where the former is labeled with Γ_k, and the latter is labeled with Ξ. We declare that C' is a subtree of C that is rooted in v_k. Then, C is a linear chase tree for Γ whose node-width is bounded by m_Γ. This concludes the induction step, and thus the proof of Lemma 4.15.

B.3 Proof of Lemma 4.16

Let us first recall the statement:

Consider a database D, and a set $\Sigma \in$ WARD of single-head TGDs. Let $\Theta \subseteq \mathsf{chase}(D, \Sigma)$ and $\Gamma \subseteq U(\mathcal{G}^{D,\Sigma}, \Theta)$. There exists a chase tree C for Γ such that $\mathsf{nwd}(C) \leq f_{\mathsf{WARD}}(\Gamma, \Sigma)$.

Let $m_\Gamma = f_{\mathsf{WARD}}(\Gamma, \Sigma)$. The proof is by induction on $\mathsf{dp}(\Gamma)$.

Base Case. Assume first that $\mathsf{dp}(\Gamma) = 1$. Then, Γ must consist of a set of facts $\{\alpha_1, \ldots, \alpha_k\} \subseteq D$, and thus a trivial chase tree for Γ is the tree that has its root labeled with Γ. The node-width of that tree is trivially at most m_Γ, as needed.

Induction Step. Suppose now that $\mathsf{dp}(\Gamma) = n + 1$. We perform a subsidiary induction on the number of atoms in Γ that have depth $n + 1$.

- Suppose first that there is exactly one atom $\alpha \in \Gamma$ that has depth $n + 1$. Let $\beta_1, \ldots, \beta_k$ be such that $\mathsf{succ}_{\sigma,h}(\alpha) = \{\beta_1, \ldots, \beta_k\}$ for some $\sigma \in \Sigma$ and some homomorphism h, and such that the application of $\beta_1, \ldots, \beta_k \Rightarrow_{\sigma,h} \alpha$ is not blocked in Γ; it is easy to see that such an application is not blocked since α is of maximal depth. Since Σ is warded, there is at most one ward β_i such that all the nulls contained in α are also present in β_i. Moreover, β_i does not share any other nulls with any of the β_j, for $j \neq i$. In case such a ward β_i exists, we set $\Gamma' = (\Gamma \setminus \{\alpha\}) \cup \{\beta_i\}$ and $\Gamma'' = \{\beta_1, \ldots, \beta_{i-1}, \beta_{i+1}, \ldots, \beta_k\}$. Otherwise, we set $\Gamma' = \Gamma \setminus \{\alpha\}$ and $\Gamma'' = \{\beta_1, \ldots, \beta_k\}$. In both cases, we see that $\{\Gamma', \Gamma''\}$ is a decomposition of Γ since the nulls that do not appear in α, yet that appear in some atom among $\beta_1, \ldots, \beta_k$, are all fresh by

Lemma B.1. Moreover, we know that there are no nulls that are present in α, yet not in any of the $\beta_1, \ldots, \beta_k$, since the application of $\beta_1, \ldots, \beta_k \Rightarrow_{\sigma,h} \alpha$ is not blocked in Γ.
Notice that we have $\mathrm{dp}(\Gamma') \leq n$ and $\mathrm{dp}(\Gamma'') \leq n$. Hence, by induction hypothesis, there are chase trees C' and C'' for Γ' and Γ'', respectively. We build a chase tree C for Γ by labeling its root v_0 with Γ, and declaring that v_0 has one child v_1 whose label is $\Gamma' \cup \Gamma''$. Furthermore, v_1 has two children, v' and v'', that are labeled with Γ' and Γ'', respectively. Notice that $m_{\Gamma'} \leq m_\Gamma$ and that $m_{\Gamma''} \leq m_\Gamma$. Moreover, $|\Gamma' \cup \Gamma''| \leq |\Gamma| + \max\{|\mathrm{body}(\sigma)| \mid \sigma \in \Sigma\} \leq m_\Gamma$. Thus, C is a chase tree for Γ with the desired bound on the node-width.

- The induction step of the subsidiary induction is performed similarly to the base case, and thus the details are omitted.

This complete the proof of Lemma 4.16.

B.4 Proof of Lemma 4.17

Let us first recall the statement:

Consider a database D and a set Σ of TGDs. Let $\Theta \subseteq \mathsf{chase}(D, \Sigma)$, $q(\bar{x})$ be a CQ, and $\bar{c}$ be a tuple of constants such that $h(\mathsf{atoms}(q)) \subseteq U(\mathcal{G}^{D,\Sigma}, \Theta)$ and $h(\bar{x}) = \bar{c}$, for some homomorphism h. If there exists a (linear) chase tree C for $h(\mathsf{atoms}(q))$ with $\mathsf{nwd}(C) \leq m$, then there exists a (linear) proof tree $\mathcal{P}$ for q w.r.t. Σ such that $\mathsf{nwd}(\mathcal{P}) \leq m$ and $\bar{c} \in \mathcal{P}(D)$.

For a sequence of variables $\bar{y}$ of $\mathsf{var}(q)$, let $\sim_{h,\bar{y}}$ be the equivalence relation defined by

$$y_i \sim_{h,\bar{y}} y_j \iff h(y_i) = h(y_j).$$

Let $\pi_{h,\bar{y}}$ be the partition of the variables $\bar{y}$ given by the set of equivalence classes of $\sim_{h,\bar{y}}$. We proceed to show that there exists a (linear) proof tree $\mathcal{P} = (\cdot, \cdot, \pi_{h,\bar{x}})$ of $q(\bar{x})$ w.r.t. Σ, i.e., the partition of the output variables of q is $\pi_{h,\bar{x}}$, with $\mathsf{nwd}(\mathcal{P}) \leq m$ and $\bar{c} \in \mathcal{P}(D)$. The proof is by induction on the depth of C (i.e., the longest among all paths that lead from the root to a leaf). In what follows, we assume that $\mathsf{atoms}(q) = \{\alpha_1, \ldots, \alpha_s\}$, and $\bar{x} = x_1, \ldots, x_n$.

Base Case. Suppose first that the depth of C is 1, i.e., C consists of a single node v_0 whose label is $h(\mathsf{atoms}(q))$. Then, the proof tree $\mathcal{P}$ that consists of a single node labeled with

$$Q(\mathsf{eq}_{\pi_{h,\bar{x}}}(\bar{x})) \leftarrow \mathsf{eq}_{\pi_{h,\bar{x}}}(\alpha_1, \ldots, \alpha_s),$$

is clearly a proof tree for $q(\bar{x})$ w.r.t. Σ such that $\bar{c} \in \mathcal{P}(D)$.

Induction Step. Suppose now that the depth of C is larger than one such that $\mathsf{nwd}(C) \leq m$. We proceed by case analysis.

Case 1. Assume that the children of the root v_0 of C, whose label is $h(\mathsf{atoms}(q))$, result from a decomposition step. Let us concentrate on the case where v_0 has exactly two children v_1 and v_2 that are labeled by Θ_1 and Θ_2, respectively; the case with more than two children is treated analogously. Clearly, $\{\Theta_1, \Theta_2\}$ must be a decomposition of $h(\mathsf{atoms}(q))$ such that $h(\mathsf{atoms}(q)) = \Theta_1 \cup \Theta_2$, and Θ_1 and Θ_2 do not share any labeled null. Let C_1 and C_2 be the subtrees rooted at v_1 and v_2, respectively, and note that $\mathsf{nwd}(C_1) \leq m$ and $\mathsf{nwd}(C_2) \leq m$. Moreover, let $\bar{y} = y_1, \ldots, y_k$ be the variables from $\mathsf{var}(q) \setminus \{x_1, \ldots, x_n\}$ such that $h(y_i)$ is a constant, and y_i occurs in atoms α and β of $q(\bar{x})$, but neither $h(\{\alpha, \beta\}) \subseteq \Theta_1$, nor $h(\{\alpha, \beta\}) \subseteq \Theta_2$ holds. Let $\mathcal{P}$ be the proof tree whose root v_0 is labeled with

$$Q(\mathsf{eq}_{\pi_{h,\bar{x}}}(\bar{x})) \leftarrow \mathsf{eq}_{\pi_{h,\bar{x}}}(\alpha_1, \ldots, \alpha_s), \tag{1}$$

and that has exactly one child v' whose label is

$$Q(\mathsf{eq}_{\pi_{h,\bar{x},\bar{y}}}(\bar{x}, \bar{y})) \leftarrow \mathsf{eq}_{\pi_{h,\bar{x},\bar{y}}}(\alpha_1, \ldots, \alpha_s). \tag{2}$$

It is easy to check that (2) results from a specialization step from (1). Now, let $\beta_{i_1}, \ldots, \beta_{i_l}$ be the atoms from $\mathrm{eq}_{\pi_{h,\bar{x},\bar{y}}}(\alpha_1, \ldots, \alpha_s)$ whose image under h is in Θ_1, and $\beta_{j_1}, \ldots, \beta_{j_r}$ those atoms whose image under h is in Θ_2. Let $\bar{z}$ be the restriction of $\mathrm{eq}_{\pi_{h,\bar{x},\bar{y}}}(\bar{x}, \bar{y})$ to $\mathrm{var}(\{\beta_{i_1}, \ldots, \beta_{i_l}\})$, and let $\bar{w}$ be the restriction of $\mathrm{eq}_{\pi_{h,\bar{x},\bar{y}}}(\bar{x}, \bar{y})$ to $\mathrm{var}(\{\beta_{j_1}, \ldots, \beta_{j_r}\})$. Let v_1 and v_2 be children of v' in $\mathcal{P}$ labeled by $Q(\bar{z}) \leftarrow \beta_{i_1}, \ldots, \beta_{i_l}$ and $Q(\bar{w}) \leftarrow \beta_{j_1}, \ldots, \beta_{j_r}$, respectively. These two queries result from a decomposition step on (2). Since $h(\{\beta_{i_1}, \ldots, \beta_{i_l}\}) = \Theta_1$ and $h(\{\beta_{j_1}, \ldots, \beta_{j_r}\}) = \Theta_2$, by induction hypothesis, there are proof trees $\mathcal{P}_1 = (\cdot, \cdot, \pi_{h,\bar{z}})$ and $\mathcal{P}_2 = (\cdot, \cdot, \pi_{h,\bar{w}})$ such that $\mathrm{nwd}(\mathcal{P}_1) \leq m$ and $\mathrm{nwd}(\mathcal{P}_2) \leq m$. Furthermore, $h(\bar{z}) \in \mathcal{P}_1(D)$ and $h(\bar{w}) \in \mathcal{P}_2(D)$. Notice that, by construction, $\mathrm{eq}_{\pi_{h,\bar{z}}}(\bar{z}) = \bar{z}$ and $\mathrm{eq}_{\pi_{h,\bar{w}}}(\bar{w}) = \bar{w}$. Moreover, $\exists \bar{v}\,(q_{\mathcal{P}_1}(\bar{z}) \wedge q_{\mathcal{P}_2}(\bar{w})) \equiv q_{\mathcal{P}}$, where $\bar{v}$ is the sequence of variables that appear in the head of (2), but not in the head of (1); we write $q_{\mathcal{P}'}$ for the CQ induced by a proof tree $\mathcal{P}'$. Hence, $\bar{c} \in q_{\mathcal{P}}(D)$, which in turn implies that $\mathcal{P}$ is the proof tree of $q(\bar{x})$ w.r.t. Σ that we are looking for.

Case 2. Suppose now that the root v_0 of C has exactly one child v' that is labeled with Θ', which results from $h(\mathrm{atoms}(q))$ by unfolding. Let $\sigma \in \Sigma$, $h_0, \beta_1, \ldots, \beta_k$, and $\alpha \in h(\mathrm{atoms}(q))$ be such that $\beta_1, \ldots, \beta_k \Rightarrow_{\sigma,h_0} \alpha$, and $\Theta' = (h(\mathrm{atoms}(q)) \setminus \{\alpha\}) \cup \{\beta_1, \ldots, \beta_k\}$. Thus, σ is of the form

$$R_{\beta_1}(\bar{x}_1), \ldots, R_{\beta_k}(\bar{x}_k) \rightarrow \exists w_{i_1}, \ldots, w_{i_l}\, R_\alpha(w_1, \ldots, w_r),$$

for some predicates $R_{\beta_1}, \ldots, R_{\beta_k}, R_\alpha$. Also, $q(\bar{x})$ contains an atom $R_\alpha(t_1, \ldots, t_r)$ such that

$$h(p_\alpha(t_1, \ldots, t_r)) \;=\; \alpha \;=\; h_0(p_\alpha(w_1, \ldots, w_r))$$

($t_1, \ldots, t_r$ are terms each of which is either a variable or a constant). Now, let σ_{v_0} be a copy of σ, where every variable occurrence x is renamed to x_{v_0}. Let h' be the homomorphism such that $h'(x_{v_0}) = h_0(x)$. Moreover, let $\mathrm{eq}_{\pi_{h,\bar{x}}}(t_1, \ldots, t_r) = s_1, \ldots, s_r$. Note that $\{t_1, \ldots, t_r\} \subseteq \{s_1, \ldots, s_r\}$, and observe that $h(R_\alpha(s_1, \ldots, s_r)) = \alpha = h'(R_\alpha(w_{1,v_0}, \ldots, w_{r,v_0}))$. Let γ be a substitution such that, for each $i, j \in [r]$,

$$\gamma(z_i) = \gamma(w_{j,v_0}) = v_t \iff t = h(z_i) = h'(w_{j,v_0}),$$

where the v_t are newly chosen variable names (for the other variables that are not mentioned, γ is simply the identity). In particular, $\gamma(x_i) = \gamma(x_j)$ iff $x_i \sim_{h,\bar{x}} x_j$, for all output variables x_i and x_j of $q(\bar{x})$ that are among $\{s_1, \ldots, s_r\}$. Let γ_0 be an MGU such that $\gamma = \eta \circ \gamma_0$ for some substitution η. Notice that if x_i and x_j are output variables of $q(\bar{x})$ among $\{s_1, \ldots, s_r\}$, then $\gamma_0(x_i) = \gamma_0(x_j)$ implies $x_i \sim_{h,\bar{x}} x_j$. On the other hand, with $\hat{x}_1, \ldots, \hat{x}_n = \mathrm{eq}_{\pi_{h,\bar{x}}}(x_1, \ldots, x_n)$, if $x_i \sim_{h,\bar{x}} x_j$, then there is exactly one $v \in \{\hat{x}_1, \ldots, \hat{x}_n\}$ such that $h(v) = h(x_i) = h(x_j)$. Thus, γ_0 is bijective when restricted to the (representatives of the) equivalence classes of $\sim_{h,\bar{x}}$, and we can henceforth assume w.l.o.g. that $\gamma_0(\hat{x}_i) = \hat{x}_i$ for each $i \in [n]$.
Let $\mathcal{P} = (\cdot, \cdot, \pi_{h,\bar{x}})$ whose root v_0 is labeled with

$$Q(\hat{x}_1, \ldots, \hat{x}_n) \;\leftarrow\; \mathrm{eq}_{\pi_{h,\bar{x}}}(\alpha_1, \ldots, \alpha_s). \tag{3}$$

We introduce a new node v' in $\mathcal{P}$ that is a child of v_0 and whose label is

$$Q(\hat{x}_1, \ldots, \hat{x}_n) \;\leftarrow\; \gamma_0(A), \tag{4}$$

where

$$A \;=\; (\mathrm{eq}_{\pi_{h,\bar{x}}}(\{\alpha_1, \ldots, \alpha_s\}) \setminus \{R_\alpha(s_1, \ldots, s_r)\}) \;\cup\; \{R_{\beta_1}(\bar{x}_{1,v_0}), \ldots, R_{\beta_k}(\bar{x}_{k,v_0})\}.$$

It is clear that (4) is a σ_{v_0}-resolvent of (3). Notice, in particular, that the variables occurring in $\mathrm{eq}_{\pi_{h,\bar{x}}}(\alpha_1, \ldots, \alpha_s)$, and unify with some existential variable from the head of σ_{v_0}, cannot be shared since the application of $\beta_1, \ldots, \beta_k \Rightarrow_{\sigma,h_0} \alpha$ is not blocked in $h(\mathrm{atoms}(q))$. Moreover, the resolvent must be IDO since γ_0 is the identity on $\{\hat{x}_1, \ldots, \hat{x}_n\}$.

Let us write $q'(\hat{x}_1, \ldots, \hat{x}_n)$ for the CQ (4). Let h'' be the homomorphism that extends h so that h'' maps q' to Θ' and $h''(\hat{x}_1, \ldots, \hat{x}_n) = \bar{c}$, which exists by construction. The subtree of C that is rooted at v', called C', has smaller depth than C. Hence, by induction hypothesis, it follows that there is a proof tree $\mathcal{P}' = (\cdot, \cdot, \pi_{h'', \hat{x}_1, \ldots, \hat{x}_n})$ of q' w.r.t. Σ such that $\mathsf{nwd}(\mathcal{P}') \leq m$, and $\bar{c} \in \mathcal{P}'(D)$. We can thus simply declare that $\mathcal{P}'$ becomes a subtree of $\mathcal{P}$ rooted at the node v' of $\mathcal{P}$. Then, $\mathcal{P} = (\cdot, \cdot, \pi_{h, \bar{x}})$ is a proof tree for $q(\bar{x})$ w.r.t. Σ, and for which it holds that $\mathsf{nwd}(\mathcal{P}) \leq m$. Moreover, we have that $\bar{c} \in \mathcal{P}(D)$ since $\mathcal{P}$ and $\mathcal{P}'$ have the same leaf nodes.

Let us finally remark that the constructions performed in the above two cases yield a linear proof tree $\mathcal{P}$ whenever C is linear. This completes the proof of Lemma 4.17.

Received ; revised ; accepted