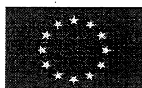
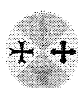




UNIVERSITA' DELLA CALABRIA

Dipartimento di INGEGNERIA INFORMATICA, MODELLISTICA, ELETTRONICA E SISTEMISTICA

Dottorato di Ricerca in

INGEGNERIA DEI SISTEMI ED INFORMATICA

Con il contributo di (Ente finanziatore)

Commissione Europea, Fondo Sociale Europeo e della Regione Calabria

CICLO

XXVIII

ON THE DECIDABILITY OF LOGIC PROGRAMS AND CHASE ALGORITHMS

Settore Scientifico Disciplinare ING-INF/05

Coordinatore: Ch.mo Prof. Felice Crupi

Firma ____________________

Supervisore/Tutor: Ch.mo Prof. Sergio Greco

Firma ____________________

Dottorando: Dott. Marco Calautti

Firma ____________________

Abstract

The problem of programs termination is a fundamental problem in Computer Science, and has always gained interest from research communities, due to the challenge of dealing with a problem that has been proved to be undecidable in general. Furthermore, there has been a great increase of interest from the logic programming and database communities in identifying meaningful and large fragments of the languages used, in order to guarantee termination of inference tasks. The goal of this thesis is to study the termination problem in the field of logic programming with function symbols and in the field of integrity database dependencies enforced via the Chase procedure. The state of the art for both fields is presented, identifying limitations of current works and new approaches to overcome such limitations are proposed.

To my family and friends

Acknowledgements*

First, I want to thank my advisor Prof. Sergio Greco, for his guidance, advices and kindness during the three years of my Ph.D course. He led me through my research field, always providing strong and inspiring ideas that allowed me to give my contribution to this exciting research field. I also acknowledge Irina Trubitsyna, Cristian Molinaro and Andreas Pieris as invaluable mentors and contributors that together with Sergio, have given shape to my research activities.

I gratefully thank my friends, for always being on my side, giving me the necessary strength and support to pursue my objectives. In particular, I am grateful to my wonderful roommates Antonio, Vincenzo and Claudio, and my friend Mara for making my time in Rende so enjoyable. I have spent some of the best years of my life with them and I am grateful to have met such amazing people. I also thank my long lasting friends Andrea, Federica, Alessandro, Rossana and Giusi, who proved that real friendship go beyond distance and time.

Lastly, I would like to thank my family, in particular my parents for their love and encouragement and my brother, who has been a friend, other than part of my family.

* La presente tesi è cofinanziata con il sostegno della Commissione Europea, Fondo Sociale Europeo e della Regione Calabria. L'autore è il solo responsabile di questa tesi e la Commissione Europea e la Regione Calabria declinano ogni responsabilità sull'uso che potrà essere fatto delle informazioni in essa contenute.

Contents

1 Introduction ... 1
- 1.1 Contributions ... 4
- 1.2 Organization ... 5

2 Preliminaries ... 6
- 2.1 Logic programs with function symbols ... 6
- 2.2 Database dependencies and the Chase ... 9
  - 2.2.1 The Chase procedure ... 12
  - 2.2.2 Computing universal models with the Chase ... 15

3 Termination of programs with function symbols ... 17
- 3.1 Limited programs ... 17
- 3.2 Analysis of limited programs ... 18
- 3.3 Termination criteria ... 19
  - 3.3.1 Ω-restriction ... 19
  - 3.3.2 λ-restriction ... 21
  - 3.3.3 Finite-domain ... 22
  - 3.3.4 Argument ranking ... 24
- 3.4 Argument-based approach: Mapping-restriction ... 26
  - 3.4.1 Complexity ... 33
- 3.5 Graph-based approaches ... 40
  - 3.5.1 Γ-acyclicity ... 41
  - 3.5.2 Safe programs ... 50
- 3.6 Constraints-based approaches ... 58
  - 3.6.1 Rule-bounded programs ... 58
  - 3.6.2 Cycle-bounded Programs ... 69

3.6.3 Complexity 74
3.6.4 Extending rule boundedness: Size-restriction 77

4 Termination of the Chase 97
4.1 The Chase termination problem 97
4.2 State of the art 99
4.2.1 Weak acyclicity 99
4.2.2 Rich Acyclicity 100
4.2.3 (C-)Stratification 100
4.2.4 Safety 101
4.2.5 Super weak acyclicity 102
4.2.6 Safe restriction and Inductive restriction 103
4.2.7 Local Stratification 104
4.2.8 Model-faithful acyclicity 105
4.2.9 Rewriting technique 105
4.3 Termination of Guarded rules 106
4.3.1 Linearity 106
4.3.2 (Weak-)Guardedness 120
4.4 Dealing with EGDs 127
4.4.1 Semi-Stratification 132
4.4.2 Adornment Algorithm 134
4.4.3 Expressivity, Complexity and Experimental Evaluation . 143

5 Conclusions 145

6 Appendix 147

References 175

1

Introduction

The problem of deciding whether a program terminates has been widely recognized as a fundamental problem in Computer Science since the '30s, when Alan Turing introduced its automatic machine (the Turing machine) and first proved that problems that cannot be solved in a finite amount of time actually exist. This has been shown by proving that the problem of checking whether a Turing machine (or any other Turing-equivalent program) always halts, for every possible input, is undecidable (i.e., it is unsolvable). Since then, researchers have deeply investigated the termination problem for programs encoded in many other languages and formalisms, with the aim to identify subclasses of such languages for which the problem becomes decidable.

One of such languages is the language of logic programs with function symbols. Function symbols are widely acknowledged as an important feature in logic programming as they make modeling easier and increase the language's expressive power, but at the same time they immediately make common reasoning tasks undecidable. Logic programs with function symbols are a form of declarative programming where problems are encoded with sets of implications (rules). Solutions are represented by some bottom-up based semantics of the underlying theory defined by the rules, such as stable models [46, 43].

Example 1.1. Consider the following simple program $\mathcal{P}_{4.51}$:

$$\mathtt{nat(s(X))} \leftarrow \mathtt{nat(X)}.$$

The program above states that if `X` is a natural number, then the *successor* of `X`, `s(X)` (where `s` denotes a function symbol), is a natural number as well. Given the fact `nat(0)`, stating that 0 is a natural number, the (unique) stable model of $\mathcal{P}_{4.51}$ is the set $\{\texttt{nat(0)}, \texttt{nat(s(0))}, \texttt{nat(s(s(0)))}, \ldots\}$. It is easy to see that whatever set of facts we add, the program does not admit a finite stable model, that is, *there is no terminating procedure* which is able to construct the stable model of this program. □

In this context, [26] introduced the class of *finitely-ground programs*, guaranteeing the existence of a finite set of stable models, each of finite size which are actually computable. However, the class is undecidable too, thus decidable subclasses have been proposed: *ω-restricted programs* [76], *λ-restricted programs* [44], *finite domain programs* [26], *argument-restricted programs* [60], A more general class is *bounded programs* [52]. An adornment-based approach that can be used in conjunction with the techniques above to detect more programs as finitely-ground has been proposed in [53].

A significant body of work has also been done on the termination of logic programs under top-down evaluation [32, 62, 68, 30, 73, 67, 15, 12, 80] and in the area of term rewriting [75, 9, 36]. Termination properties of query evaluation for normal programs under tabling have been studied in [71, 72, 79]. Another approach are $\mathbb{FDNC}$ programs [35], which have infinite answer sets in general, but a finite representation that can be exploited for knowledge compilation and fast query answering.

But, unfortunately, all the works above cannot be straightforwardly applied to the setting of logic programs evaluated in a bottom-up fashion—for a discussion on this see, e.g., [26, 6].

Another prominent class of programs for which termination is of particular interest is the class of programs defined via logical constraints, used in the field of database theory to enforce some desirable properties over the underlying database. Satisfaction of such constraints is enforced via the well-known Chase procedure.

The Chase is a well-known algorithm originally proposed for classical database problems, such as query optimization, query containment and equivalence, dependency implication, and database schema design [4, 13, 51, 61]. In recent years, it has seen a revival of interest because of a wide range of applications where it plays a central role, such as data exchange, data cleaning and repairing, data integration, and ontological reasoning [38, 14, 8, 7, 24, 31, 45, 40].

The execution of the Chase involves the enforcement of two kind of logical constraints: *tuple generating* and *equality generating* dependencies. The Chase will insert tuples possibly with null values to satisfy a tuple generating dependency (TGD), and will replace null values with constants or other null values to satisfy an equality generating dependency (EGD). Specifically, the Chase consists of applying a sequence of steps, where each step enforces a dependency that is not satisfied by the current instance. It might well be the case that multiple dependencies can be enforced and, in this case, the Chase picks one nondeterministically. Different choices lead to different sequences, some of which might be terminating, while others might not. This aspect is illustrated in the following example.

Example 1.2. Consider the set of dependencies $\Sigma_{1.2}$ below:

$$
\begin{aligned}
r_1 &: \; n(X) \rightarrow \exists Y \; e(X,Y) \\
r_2 &: \; e(X,Y) \rightarrow n(Y) \\
r_3 &: \; e(X,Y) \rightarrow X = Y
\end{aligned}
$$

and the database $D = \{n(a)\}$. All dependencies are satisfied by D, except for r_1. Thus, the Chase enforces r_1 by adding $e(a, z_1)$ to D, where z_1 is a (labeled) null value. However, this causes both r_2 and r_3 to be violated: r_2 requires the fact $n(z_1)$, while r_3 says that a and z_1 should be the same. Suppose the Chase chooses to enforce r_2, and thus $n(z_1)$ is added to the current instance. Now r_1 is not satisfied again, while r_3 continues to be violated. Suppose the Chase chooses to enforce r_1. Then, similar to the first step, $e(z_1, z_2)$ is added to the current instance, and this causes r_2 to become violated again. It is easy to see that repeatedly enforcing first r_1 and then r_2 yields an infinite Chase sequence that introduces an infinite number of facts: $n(z_2), e(z_2, z_3), n(z_3), ...$

However, by enforcing first r_1 and then r_3, we get a terminating Chase sequence. Specifically, enforcing r_1 adds $e(a, z_1)$ to D. Then, the application of r_3 updates the null value z_1 to a. At this point, no further dependency needs to be enforced, and the Chase terminates with the resulting database being $\{n(a), e(a, a)\}$. □

The importance of the Chase in many applications is due to the fact that several problems (e.g., checking query containment under dependencies, checking implication of dependencies, computing solutions in data exchange, and computing certain answers in data integration) can be solved by exhibiting a *universal model*, and the Chase computes a universal model, when it termi-

nates [33]. Roughy speaking, a *model* for a database and a set of dependencies is a finite instance that includes the database and satisfies the dependencies. A *universal model* is a model that can be "mapped" to every other model—in a sense, it represents the entire space of possible models (formal definitions are reported in the preliminaries). Universal models are slight generalizations of universal solutions in the data exchange setting [38], and can be used to compute them. Moreover, the certain answers to a conjunctive query in the presence of dependencies can be computed by evaluating the query over a universal model (rather than considering all models). Other applications of universal models (e.g., dependency implication and query containment under dependencies) can be found in [33].

Thus, finding a universal model is a central problem in many applications and, once again, the Chase is a tool to solve it, provided that it terminates. As a consequence, checking whether the Chase terminates becomes a central problem, but at the same time it is an undecidable one [33, 49, 47].

1.1 Contributions

The contribution of this thesis is to formally introduce the termination problem for logic programs with function symbols and for database dependencies; present the state of the art for both fields, identifying limitations of current works and propose new approaches to overcome such limitations by presenting the results published in [18, 17, 16, 21, 19, 20]. In particular, for logic programs with function symbols, first the works from [18, 17] are presented, in order to show how most of the techniques in the literature are not able to actually exploit the role of function symbols in logic programs, in order to understand that a given program is terminating, and show a way to actually deal with function symbols in logic programs.

Then, the works from [16, 21] are presented, showing that current approaches, including the ones from [18, 17] still perform a limited analysis of how values, propagated by means of rules, influence each other.

For the case of database dependencies, the work from [19] is discussed as the first known effort to identify fragments of TGDs for which the termination of the Chase is decidable. Finally, the work from [20] is presented as the first one able to directly deal with the Chase termination problem in the presence of both TGDs and EGDs.

1.2 Organization

The thesis is organized as follows. Chapter 2 formally introduces syntax and semantics of logic programs with function symbols and database dependencies, along with the definition of the Chase algorithm and its variations. Chapter 3 discusses the termination problem for logic programs with function symbols, currently known works and present the three contributions in this research field. Chapter 4 presents the Chase termination problem, and how the problem of the termination of the Chase changes when considering different variation of the Chase. Furthermore, termination criteria known in the literature are discussed and two contributions are introduced. Finally Chapter 5 is devoted to conclusions and future work. All results regarding the author's contributions have inline proofs, except for results presented in Sections 4.3 and 4.4. Due to their length, the proofs required for these sections are included in the appendix.

2

Preliminaries

This chapter covers notions and notations used in this thesis. We will first recall syntax and semantics for logic programs with functions symbols. Then syntax and semantics of database dependencies are introduced along with the Chase procedure and its variations.

2.1 Logic programs with function symbols

We assume to have (pairwise disjoint) infinite sets of *logical variables*, *predicate symbols*, and *function symbols*. Logical variables are used in logic programs and are denoted by upper-case letters.

Each predicate and function symbol g is associated with an *arity* $arity(g)$, which is a non-negative integer. Function symbols of arity 0 are called *constants*. The set of function symbols occurring in a program $\mathcal{P}$ is denoted by $F_{\mathcal{P}}$. Furthermore, $F_{\mathcal{P}}^*$ denotes the set of all strings plus the empty string (denoted by ϵ) constructed by using the function symbols in $F_{\mathcal{P}}$.

A *term* t is either a logical variable or an expression of the form $f(t_1, ..., t_m)$, where f is a function symbol of arity $m \geq 0$ and $t_1, ..., t_m$ are terms; if $m = 0$, t is said to be a *constant*. Logical variables and constants are said to be *simple* terms, whereas all other terms are said to be *complex*. We use $\bar{t}$ to denote sequences of terms.

The binary relation *subterm* over terms is recursively defined as follows: every term is a subterm of itself; if t is a complex term of the form $f(t_1, ..., t_m)$, then every t_i is a subterm of t for $1 \leq i \leq m$; if t_1 is a subterm of t_2 and t_2 is a subterm of t_3, then t_1 is a subterm of t_3. The depth $d(u, t)$ of a simple term u in a term t that contains u is recursively defined as follows:

$$d(u,u) = 0,$$
$$d(u, f(t_1, ..., t_m)) = 1 + \max_{i \,:\, t_i \, contains \, u} d(u, t_i).$$

The *depth of term* t, denoted by $d(t)$, is the maximal depth of all simple terms occurring in t.

An *atom* is of the form $p(t_1, ..., t_n)$, where p is a predicate symbol of arity $n \geq 0$ and $t_1, ..., t_n$ are terms—we also call the atom a p-atom. We use $pr(A)$ to denote the predicate symbol (resp. set of predicate symbols) of an atom A (resp set of atoms). A *literal* is either an atom A (*positive* literal) or its negation $\neg A$ (*negative* literal).

A *rule* r is of the form:

$$A_1 \vee ... \vee A_m \;\leftarrow\; B_1, ..., B_k, \neg C_1, ..., \neg C_n$$

where $m > 0$, $k \geq 0$, $n \geq 0$, and $A_1, ..., A_m, B_1, ..., B_k,\ C_1,\ ..., C_n$ are atoms. The disjunction $A_1 \vee ... \vee A_m$ is called the *head* of r and is denoted by $head(r)$. The conjunction $B_1, ..., B_k, \neg C_1, ..., \neg C_n$ is called the *body* of r and is denoted by $body(r)$. Furthermore, the *positive body of* r, denoted as $body^+(r)$ is the conjunction $B_1, ..., B_k$. With a slight abuse of notation, we sometimes use $body(r)$ (resp. $body^+(r)$, $head(r)$) to also denote the *set* of literals appearing in the body (resp. positive body, head) of r. If $m = 1$, then r is *normal*; in this case, $head(r)$ denotes the head atom. If $n = 0$, then r is *positive*.

A *program* is a finite set of rules. A program is *normal* (resp. *positive*) if every rule in it is normal (resp. positive). We assume that programs are *range restricted*, i.e., for every rule, every logical variable appears in some positive body literal. A term (resp. atom, literal, rule, program) is *ground* if no logical variables occur in it. A ground normal rule with an empty body is also called a *fact*.

Let $\mathcal{P}$ be a program. The set of all predicate symbols appearing in $\mathcal{P}$ (resp. appearing in the head of a rule in $\mathcal{P}$) is denoted as $pred(\mathcal{P})$ (resp. $def(\mathcal{P})$). A predicate symbol p depends on a predicate q if there is a rule r in $\mathcal{P}$ such that p appears in the head and q in the body, or there is a predicate s such that p depends on s and s depends on q. A predicate p is said to be *recursive* if it depends on itself, whereas two predicates p and q are said to be *mutually recursive* if p depends on q and q depends on p.

Given a predicate symbol p of arity n, the i-th *argument* of p is an expression of the form $p[i]$, for $1 \leq i \leq n$. Given a set S of predicate symbols,

$args(S)$ denotes the set of all arguments of the predicate symbols in S. With a slight abuse of notation ,for a program $\mathcal{P}$, $args(\mathcal{P})$ denotes $args(pred(\mathcal{P}))$.

A *substitution* θ is of the form $\{X_1/t_1, ..., X_n/t_n\}$, where $X_1, ..., X_n$ are distinct logical variables and $t_1, ..., t_n$ are terms. The result of applying θ to an atom (or term) A, denoted $A\theta$, is the atom (or term) obtained from A by simultaneously replacing each occurrence of a logical variable X_i in A with t_i if X_i/t_i belongs to θ. Two atoms A_1 and A_2 *unify* if there exists a substitution θ, called a *unifier* of A_1 and A_2, such that $A_1\theta = A_2\theta$. The *composition* of two substitutions $\theta = \{X_1/t_1, ..., X_n/t_n\}$ and $\vartheta = \{Y_1/u_1, ..., Y_m/u_m\}$, denoted $\theta \circ \vartheta$, is the substitution obtained from the set $\{X_1/t_1\vartheta, ..., X_n/t_n\vartheta, Y_1/u_1, ..., Y_m/u_m\}$ by removing every $X_i/t_i\vartheta$ such that $X_i = t_i\vartheta$ and every Y_j/u_j such that $Y_j \in \{X_1, ..., X_n\}$. A substitution θ is *more general* than a substitution ϑ if there exists a substitution η such that $\vartheta = \theta \circ \eta$. A unifier θ of A_1 and A_2 is called a *most general unifier* (mgu) of A_1 and A_2 if it is more general than any other unifier of A_1 and A_2 (indeed, the mgu is unique modulo renaming of logical variables).

Consider a program $\mathcal{P}$. The *Herbrand universe* $H_{\mathcal{P}}$ of $\mathcal{P}$ is the possibly infinite set of ground terms constructible using function symbols (and thus, also constants) appearing in $\mathcal{P}$. The *Herbrand base* $B_{\mathcal{P}}$ of $\mathcal{P}$ is the set of ground atoms constructible using predicate symbols appearing in $\mathcal{P}$ and ground terms of $H_{\mathcal{P}}$.

A rule (resp. atom) r' is a *ground instance* of a rule (resp. atom) r in $\mathcal{P}$ if r' can be obtained from r by substituting every logical variable in r with some ground term in $H_{\mathcal{P}}$. We use $ground(r)$ to denote the set of all ground instances of r and define $ground(\mathcal{P})$ to denote the set of all ground instances of the rules in $\mathcal{P}$, i.e., $ground(\mathcal{P}) = \cup_{r \in \mathcal{P}} ground(r)$.

An *interpretation* of $\mathcal{P}$ is any subset I of $B_{\mathcal{P}}$. The truth value of a ground atom A w.r.t. I, denoted $value_I(A)$, is *true* if $A \in I$, *false* otherwise. The truth value of $\neg A$ w.r.t. I, denoted $value_I(\neg A)$, is *true* if $A \notin I$, *false* otherwise. A ground rule r is *satisfied* by I, denoted $I \models r$, if there is a ground literal L in $body(r)$ s.t. $value_I(L) = false$ or there is a ground atom A in $head(r)$ s.t. $value_I(A) = true$. Thus, if the body of r is empty, r is satisfied by I if there is an atom A in $head(r)$ s.t. $value_I(A) = true$. An interpretation of $\mathcal{P}$ is a *model* of $\mathcal{P}$ if it satisfies every ground rule in $ground(\mathcal{P})$. A model M of $\mathcal{P}$ is minimal if no proper subset of M is a model of $\mathcal{P}$. The set of minimal models of $\mathcal{P}$ is denoted by $\mathcal{MM}(\mathcal{P})$.

Given an interpretation I of $\mathcal{P}$, let $\mathcal{P}^I$ denote the ground positive program derived from *ground*($\mathcal{P}$) by *(i)* removing every rule containing a negative literal $\neg A$ in the body with $A \in I$, and *(ii)* removing all negative literals from the remaining rules. An interpretation I is a *stable model* of $\mathcal{P}$ if $I \in \mathcal{MM}(\mathcal{P}^I)$. The set of stable models of $\mathcal{P}$ is denoted by $\mathcal{SM}(\mathcal{P})$. It is well known that stable models are minimal models (i.e., $\mathcal{SM}(\mathcal{P}) \subseteq \mathcal{MM}(\mathcal{P})$), and $\mathcal{SM}(\mathcal{P}) = \mathcal{MM}(\mathcal{P})$ for positive programs.

A positive normal program $\mathcal{P}$ has a unique minimal model, which, with a slight abuse of notation, we denote as $\mathcal{MM}(\mathcal{P})$. The *immediate consequence operator* of $\mathcal{P}$ is a function $T_\mathcal{P} : 2^{B_\mathcal{P}} \to 2^{B_\mathcal{P}}$ defined as follows: for every interpretation I, $T_\mathcal{P}(I) = \{A \mid A \leftarrow B_1, ..., B_n \in ground(\mathcal{P})$ and $\{B_1, ..., B_n\} \subseteq I\}$. The i-th iteration of $T_\mathcal{P}$ $(i \geq 1)$ w.r.t. an interpretation I is defined as follows: $T^1_\mathcal{P}(I) = T_\mathcal{P}(I)$ and $T^i_\mathcal{P}(I) = T_\mathcal{P}(T^{i-1}_\mathcal{P}(I))$ for $i > 1$. The minimal model of $\mathcal{P}$ coincides with $T^\infty_\mathcal{P}(\varnothing)$.

2.2 Database dependencies and the Chase

We define the following pairwise disjoint sets of symbols: a set $\mathbf{C}$ of *constants* (constitute the "normal" domain of a database), a set $\mathbf{N}$ of *(labeled) nulls* (used as placeholders for unknown values, and thus can be also seen as (globally) existentially quantified variables), and a set $\mathbf{V}$ of (regular) *variables* (used in dependencies). A fixed lexicographic order is assumed on $(\mathbf{C} \cup \mathbf{N})$ such that every null of $\mathbf{N}$ follows all constants of $\mathbf{C}$. We denote by $\mathbf{X}$ sequences (or sets, with a slight abuse of notation) of variables or constants $X_1, \ldots, X_k$, with $k \geqslant 0$. Throughout, let $[n] = \{1, \ldots, n\}$, for any integer $n \geqslant 1$.

A *(relational) schema* $\mathcal{R}$ is a (finite) set of *relational symbols* (or *predicates*), each with its associated arity *arity*(p). We write p/n to denote that p is an n-ary predicate. A *position* $p[i]$ (in a schema $\mathcal{R}$) is identified by a predicate $p \in \mathcal{R}$ and its i-th argument (or attribute). The set of positions of $\mathcal{R}$, denoted by *pos*($\mathcal{R}$), is defined as $\{p[i] \mid p/n \in \mathcal{R}$ and $i \in [n]\}$. A *term* t is a constant, null or variable. An *atomic formula* (or simply *atom*) has the form $p(\mathbf{t})$, where p is a relation, and $\mathbf{t}$ a tuple of terms. An atom is called a *fact* if all its terms belong to $\mathbf{C} \cup \mathbf{N}$ and *ground* if all of its terms are constants of $\mathbf{C}$. For an atom A, we refer to its predicate by *pred*(A), and we denote by *const*(A), *null*(A), *var*(A) and *pos*(A) the set of its constants, nulls, variables, and the set of its positions, respectively. Furthermore, we denote *dom*(A) = *const*(A) $\cup$ *null*(A) $\cup$ *var*(A). Given a set of positions Π, we

denote by $var(A, \Pi)$ the set of variables occurring in A at positions of Π. Furthermore, given a set of variables $\mathbf{U}$, $pos(A, \mathbf{U})$ is the set of positions in A at which variables of $\mathbf{U}$ occur. The above notations naturally extend to sets of atoms. Conjunctions of atoms are often identified with the sets of their atoms. An *instance* I is a (possibly infinite) set of atoms of the form $p(\mathbf{t})$, where $\mathbf{t}$ is a tuple of constants and nulls. A *database* D is a finite instance such that $dom(D) \subset \mathbf{C}$.

A *substitution* from a set of symbols S to a set of symbols S' is a function $h : S \to S'$ defined as follows: $\varnothing$ is a substitution (empty substitution), and if h is a substitution, then $(h \cup \{s \to s'\})$ is a substitution, where $(s, s') \in S \times S'$. The *restriction* of h to $T \subseteq S$, denoted as $h_{|T}$, is the substitution $h' = \{t \to h(t) \mid t \in T\}$. A *homomorphism* from a set of atoms A to a set of atoms A' is a substitution $h : (\mathbf{C} \cup \mathbf{N} \cup \mathbf{V}) \to (\mathbf{C} \cup \mathbf{N} \cup \mathbf{V})$ such that: $t \in \mathbf{C}$ implies $h(t) = t$, and $r(t_1, \ldots, t_n) \in A$ implies $h(r(t_1, \ldots, t_n)) = r(h(t_1), \ldots, h(t_n)) \in A'$.

Example 2.1. Consider the database $D = K_1 = \{n(a)\}$ and the set of dependencies $\Sigma_{1.2}$ of Example 1.2.

Let $h_1 : dom(body(r_1)) \to dom(K_1)$ be defined as follows: $h_1(X) = a$. Clearly, h_1 is a homomorphism from the body of r_1 to K_1. Consider now the instance $K_2 = \{n(a), e(a, z_1)\}$, and let h_2 be the mapping defined as follows: $h_2(X) = a$ and $h_2(Y) = z_1$. It is easy to see that h_2 is a homomorphism from the body of r_2 to K_2. Moreover, h_2 is also a homomorphism from the body of r_3 to K_2. □

Tuple generating dependencies

A *tuple-generating dependency (TGD)* r is a first-order formula

$$\forall \mathbf{X} \forall \mathbf{Y} (\varphi(\mathbf{X}, \mathbf{Y}) \to \exists \mathbf{Z}\, \psi(\mathbf{X}, \mathbf{Z}))$$

where $(\mathbf{X} \cup \mathbf{Y} \cup \mathbf{Z}) \subset \mathbf{V}$, and φ, ψ are conjunctions of atoms; $\varphi(\mathbf{X}, \mathbf{Y})$ is the *body* of r, denoted $body(r)$, while $\psi(\mathbf{X}, \mathbf{Z})$ is the *head* of r, denoted $head(r)$. With a slight abuse of notation, we sometimes treat $body(r)$ and $head(r)$ as sets (of atoms). A TGD is said to be *universally quantified* or *full* if all its variables are universally quantified (i.e., $\mathbf{Z}$ is empty), otherwise it is *existentially quantified*. The *frontier* of r, denoted $fr(r)$, is the set of variables $\mathbf{X}$, and we define $frpos(r)$ as the set of positions $pos(head(r), fr(r))$. Let also $ex(r) = \mathbf{Z}$. Assuming that $head(r) = A_1, \ldots, A_k$, let (r, i), where $i \in [k]$, be the single-head TGD $body(r) \to A_i$.

A TGD r is *guarded* if there exists an atom $A \in \mathit{body}(r)$ that contains (or "guards") all the variables of $\mathit{body}(r)$ [22]. The class of guarded TGDs, denoted G, is defined as the family of all possible sets of guarded TGDs. *Weakly-guarded* TGDs extend guarded TGDs by requiring only the body-variables that appear at affected positions, i.e., positions that can host nulls during the Chase, to appear in the guard; for the formal inductive definition of affected positions see [22]. The corresponding class is denoted WG. We write $\mathit{guard}(r)$ for the guard of a (weakly-)guarded TGD r. A key subclass of G are the so-called *linear* TGDs [23], that is, TGDs with just one body-atom (which is automatically a guard), and the corresponding class is denoted L. A set of linear TGDs is called *simple* if there is no repetition of variables in the body of the TGDs, and the corresponding class is denoted SL. It is straightforward to verify that $\mathsf{SL} \subset \mathsf{L} \subset \mathsf{G} \subset \mathsf{WG}$.

Equality generating dependencies

An *equality generating dependency* (EGD) is a (universally quantified) formula of the form:

$$\forall \mathbf{X} \forall \mathbf{Y}\ \varphi(\mathbf{X}, \mathbf{Y}) \rightarrow X_1 = X_2$$

where $\mathbf{X} = X_1, X_2$, $\mathbf{Y}$ is a list of variables, $\varphi(\mathbf{X}, \mathbf{Y})$ is a conjunction of atoms.

The schema of a set Σ of TGDs and EGDs, denoted $\mathit{sch}(\Sigma)$, is defined as the set of predicates occurring in Σ. Furthermore, we also denote $\mathit{pos}(\Sigma) = \mathit{pos}(\mathit{sch}(\Sigma))$.

Universal Models

Given a database D and a set of dependencies Σ, a *model* of (D, Σ) is a finite instance J such that $D \subseteq J$ and $J \models \Sigma$ (i.e., J satisfies all dependencies in Σ in the standard first-order manner). A *universal model* of (D, Σ) is a model J of (D, Σ) such that for every model J' of (D, Σ) there exists a homomorphism from J to J'. The set of all models (resp. universal models) of (D, Σ) will be denoted by $\mathit{Mod}(D, \Sigma)$ (resp. $\mathit{UMod}(D, \Sigma)$).

Example 2.2. Consider the set of dependencies $\Sigma_{2.2}$ below:

$$r_1 :\ p(X, Y) \rightarrow \exists Z\ e(X, Z)$$
$$r_2 :\ q(X, Y) \rightarrow \exists Z\ e(Z, Y)$$

and the database $D = \{p(a,b), p(c,d)\}$. Both $J_1 = D \cup \{e(a,z_1), e(z_2,d)\}$ and $J_2 = D \cup \{e(a,d)\}$ are models of $(D, \Sigma_{2.2})$. It can be shown that J_1 is a universal model, while J_2 is not. Notice that an homomorphism from J_1 to J_2 is the mapping h defined as follows: $h(z_1) = d$ and $h(z_2) = a$. In a sense, J_2 makes the somehow arbitrary assumption that the two facts required by the two TGDs are the same fact $e(a,d)$, which is not part of the specification. □

As discussed below, computing certain query answers is one of many applications where universal models play an important role, and their computation is a central problem. Consider an instance J and a query Q. Then, *(i)* $J_{\downarrow}$ denotes the set of facts in J that do not contain labeled nulls, and *(ii)* $Q(J)$ denotes the result of evaluating Q over J.

The certain answers to a query Q over a database D and a set of dependencies Σ are defined as $certain(Q, D, \Sigma) = \bigcap\{Q(J) \mid J \in Mod(D, \Sigma)\}$. The certain answers to a union of conjunctive queries Q can be computed by evaluating Q over an arbitrary universal model, that is, $certain(Q, D, \Sigma) = Q(I)_{\downarrow}$, where $I \in UMod(D, \Sigma)$. This means that to determine the certain answers to a union of conjunctive queries Q over a database D with dependencies Σ, it is not necessary to compute all models of (D, Σ), but it suffices to compute just an arbitrary universal model. Therefore, the computation of a universal model is particularly relevant. It is worth mentioning that the aforementioned property has applications in query answering under dependencies, query answering in data exchange, and query answering with incomplete and inconsistent data [38, 22].

2.2.1 The Chase procedure

When focusing on TGDs and EGDs, an algorithm for computing universal models does exist. The *Chase* takes as input a database D and a set Σ of TGDs and EGDs, and whenever it terminates without failing, it constructs a universal model of (D, Σ) [33, 38].

Below we define a *Chase step*, which consists of enforcing a TGD or an EGD. As detailed later, the Chase step is used by different variants of the Chase (standard, oblivious, semi-oblivious), each of which relies on a different condition of "applicability" of the Chase step. Thus, the following definition does not incorporate a notion of applicability, but it will be combined with different notions of applicability to define the different variants of the Chase.

In the following we consider a particular substitution, denoted by γ as either the empty set or a singleton $\{z/t\}$, where z is a labeled null and t is

either a labeled null or a constant. The result of applying γ to an expression F (e.g., term, atom, set of atoms, etc.), denoted by $F\,\gamma$, is F if $\gamma = \varnothing$, otherwise it is the expression obtained from F by replacing every occurrence of z with t.

Definition 2.3 (Chase step). *Let K be an instance, r a TGD or EGD, and h a homomorphism from body(r) to K. An expression of the form $K \xrightarrow{r,h,\gamma} J$ is a* Chase step *if the following conditions hold.*

1. *If r is a TGD $\varphi(\mathbf{X}, \mathbf{Y}) \rightarrow \exists \mathbf{Z}\, \psi(\mathbf{X}, \mathbf{Z})$ then let h' be the homomorphism obtained by extending h so that each variable in $\mathbf{Z}$ is assigned a fresh labeled null not occurring in K. Then, $J = K \cup h'(\psi(\mathbf{X}, \mathbf{Z}))$. Furthermore, γ is the empty substitution.*
2. *If r is an EGD $\varphi(\mathbf{X}, \mathbf{Y}) \rightarrow X_1 = X_2$ then $h(X_1) \neq h(X_2)$. Furthermore,*
   (a) *If $h(X_1), h(X_2) \in const$, then $J =\perp$ and γ is the empty substitution.*
   (b) *Otherwise, γ and J are defined as follows. If $h(X_1)$ is a labeled null, then $\gamma = \{h(X_1)/h(X_2)\}$; otherwise, $\gamma = \{h(X_2)/h(X_1)\}$. Moreover, $J = K\,\gamma$.* □

In a Chase step, the pair (r, h) is called *trigger* and γ is used to keep track of the substitution performed when an EGD is enforced. Thus, when considering sets Σ of TGDs only we will omit the empty substitution γ for Chase steps.

Example 2.4. Consider again the database $D = K_1 = \{n(a)\}$ and the set of dependencies $\Sigma_{1.2}$ of Example 1.2. Let h_1 be the homomorphism of Example 2.1. Then, $K_1 \xrightarrow{r_1,h_1,\gamma_1} K_2$ is a Chase step, where $K_2 = K_1 \cup \{e(a, z_1)\} = \{N(a), e(a, z_1)\}$ and γ_1 is the empty substitution (as r_1 is a TGD). Consider now the homomorphism h_2 of Example 2.1. Then, $K_2 \xrightarrow{r_2,h_2,\gamma_2} K_3$ is a Chase step, where $K_3 = K_2 \cup \{N(z_1)\} = \{n(a),\, e(a, z_1),\, n(z_1)\}$ and γ_3 is the empty substitution. Another possible Chase step starting from K_2 is $K_2 \xrightarrow{r_3,h_2,\gamma_2'} K_3'$, where $\gamma_2' = \{z_1/a\}$ and $K_3' = K_2\,\gamma_2' = \{n(a), e(a, a)\}$. □

A *Chase sequence* of (D, Σ) is a (possibly infinite) sequence of Chase steps $S = K_1 \xrightarrow{r_1,h_1,\gamma_1} K_2 \xrightarrow{r_2,h_2,\gamma_2} K_3 \cdots$ such that $K_1 = D$ and every $r_i \in \Sigma$.

Standard Chase S is a *standard Chase sequence* if it is an exhaustive application of Chase steps s.t. for each $K_i \xrightarrow{r_i,h_i,\gamma_i} K_{i+1}$ in S, if r_i is a TGD, then there is no extension of h_i to a homomorphism h_i' from $body(r_i) \cup head(r_i)$ to

K_i.

Oblivious Chase S is an *oblivious Chase sequence* if it is an exhaustive application of Chase steps s.t. for each $K_i \stackrel{r_i,h_i,\gamma_i}{\longrightarrow} K_{i+1}$ in S, there is no Chase step $K_j \stackrel{r_j,h_j,\gamma_j}{\longrightarrow} K_{j+1}$ in S such that $j < i$, $r_j = r_i = r$, and for each variable X occurring in the body of r we have that $h_i(X) = h_j(X)\,\gamma_j \,\cdots\, \gamma_{i-1}$.

Semi-Oblivious Chase S is a *semi-oblivious Chase sequence* if it is an exhaustive application of Chase steps s.t. for each $K_i \stackrel{r_i,h_i,\gamma_i}{\longrightarrow} K_{i+1}$ in S, there is no Chase step $K_j \stackrel{r_j,h_j,\gamma_j}{\longrightarrow} K_{j+1}$ in S such that $j < i$, $r_j = r_i = r$, and for each variable x occurring in $fr(r)$ we have that $h_i(X) = h_j(X)\,\gamma_j \,\cdots\, \gamma_{i-1}$.

Note that in the definitions above, when only TGDs are considered, the conditions $h_i(X) = h_j(X)\,\gamma_j \,\cdots\, \gamma_{i-1}$ for every variable X in $body(r)$ and every variable in $fr(r)$, respectively, become as follows. Oblivious case: for every variable X in $body(r)$, $h_i(X) = h_j(X)$ ($h_i = h_j$ for short): semi-oblivious case: for every variable X in $fr(r)$, $h_i(X) = h_j(X)$ ($h_i \sim_r h_j$ for short).

Example 2.5. Consider again the database $D = \{n(a)\}$ and the set of dependencies $\Sigma_{1.2}$ of Example 1.2. A standard Chase sequence of D with $\Sigma_{1.2}$ is $K_1 \stackrel{r_1,h_1,\gamma_1}{\longrightarrow} K_2 \stackrel{r_3,h_2,\gamma_2'}{\longrightarrow} K_3'$, where $K_1 = D$ and $h_1, h_2, \gamma_1, \gamma_2', K_2, K_3'$ are those reported in Example 2.4. Notice that no further Chase steps can be added to the sequence.

As mentioned in Example 1.2, another standard Chase sequence of D with $\Sigma_{1.2}$ is the (infinite) one obtained by repeatedly enforcing r_1 first and then r_2, that is $K_1 \stackrel{r_1,h_1,\gamma_1}{\longrightarrow} K_2 \stackrel{r_2,h_2,\gamma_2}{\longrightarrow} K_3 ...$, where $h_1, h_2, \gamma_1, \gamma_2, K_2$, and K_3 are those reported in Example 2.4. □

The following example shows the different behaviors of standard, oblivious, and semi-oblivious Chase sequences.

Example 2.6. Consider the database $D = K_1 = \{e(a,b)\}$ and a set $\Sigma_{2.6}$ consisting only of the following TGD r:

$$e(X,Y) \rightarrow \exists Z\ e(X,Z)$$

Since $D \models r$, the only standard Chase sequence of D with Σ is the empty sequence.

A non-empty (terminating) semi-oblivious Chase sequence is $K_1 \xrightarrow{r,h_1,\gamma_1} K_2$, where $h_1(X) = a$, $h_1(Y) = b$, γ_1 is the empty substitution, and $K_2 = K_1 \cup \{e(a, z_1)\} = \{e(a, b), e(a, z_1)\}$. Notice that adding the Chase step $K_2 \xrightarrow{r,h_2,\gamma_2} K_3$, with $h_2(X) = a$, $h_2(Y) = z_1$, $\gamma_2 = \emptyset$, and $K_3 = K_2 \cup \{e(a, z_2)\}$, does not result in a semi-oblivious Chase sequence, because of the presence of the Chase step $K_1 \xrightarrow{r,h_1,\gamma_1} K_2$ in the same Chase sequence, with $h_1(X)\gamma_1 = h_2(X) = a$.

As for the oblivious Chase, the infinite sequence whose first step is $K_1 \xrightarrow{r,h_1,\gamma_1} K_2$ discussed above, and the i-th Chase step ($i > 1$) is $K_i \xrightarrow{r,h_i,\gamma_i} K_{i+1}$, with $h_i(X) = a$, $h_i(Y) = z_{i-1}$, $\gamma_i = \emptyset$, and $K_{i+1} = K_i \cup \{e(a, z_i)\}$ is an (infinite) oblivious Chase sequence. □

2.2.2 Computing universal models with the Chase

A standard (resp. oblivious, semi-oblivious) Chase sequence S can be finite (when no further Chase step can be applied) or infinite (when there is always a further Chase step that can be applied)—in the former case we also say that the sequence is *terminating*. If S is finite and consists of m Chase steps, we say that K_m is the *result* of S. If $K_m = \perp$ then S is *failing*, otherwise it is *successful*. For instance, the first standard Chase sequence discussed in Example 2.5 is terminating, successful, and its result is K'_3. The second standard Chase sequence in Example 2.5 is not terminating.

In the presence of TGDs only, the oblivious (resp. semi-oblivious) Chase procedure is equivalent to the computation of the fixpoint of a particular Skolemized version of Σ with D, where Skolemized terms are used in place of labeled nulls. For instance, the *skolemized version* of dependency r in Example 2.6 *for the oblivious (resp., semi-oblivious) Chase* is $e(X, Y) \rightarrow e(X, f^r_Z(X, Y))$ (resp., $e(X, Y) \rightarrow e(X, f^r_Z(X))$).

It is well-known that for every database D and set of TGDs and EGDs Σ, (1) if J is the result of some successful terminating (standard, (semi-)oblivious) Chase sequence of D with Σ, then J is a universal model of (D, Σ), called *canonical*; (2) if some failing Chase sequence of D with Σ exists, then there is no model of (D, Σ). We use $\mathit{CMod}(D, \Sigma)$ to denote the set of all canonical models of (D, Σ). In some cases, we cannot produce a universal model by the Chase as there is no terminating sequence, although a model does exist.

Core Chase The *core Chase* has been proposed to identify a preferable universal model [33, 39]. To define the core Chase, we first need to introduce the notion of a core of an instance. Roughly speaking, the core of an instance J is the smallest subset of J that is also a homomorphic image of J. More

precisely, a subset C of an instance J is a *core of* J if there is a homomorphism from J to C, but there is no homomorphism from J to a proper subset of C. Cores of J are unique up to isomorphism and therefore we can talk about "the" core of J, which is denoted as $core(J)$.

A *core Chase sequence* is a sequence of *core Chase steps*. Roughly speaking, a core Chase step first applies all possible standard Chase steps "in parallel", and then computes the core of the resulting instance. As all standard Chase steps are applied in parallel, the core Chase eliminates the nondeterminism of the standard Chase. More formally, given an instance K and a set of dependencies Σ, a core Chase step consists of the following two sub-steps: *(i)* $J = \cup_{K \xrightarrow{r,h,\gamma} K'} K'$, where each $K \xrightarrow{r,h,\gamma} K'$ is understood to be a standard Chase step; *(ii)* $J' = core(J)$. Then, J' is the result of the core Chase step. [33] showed that whenever there is a universal model of (D, Σ), the core Chase is able to construct one, that is, the core Chase is a *complete* procedure for finding universal models. Moreover, every core Chase sequence of D with Σ constructs the same (up to isomorphism) universal model.

Example 2.7. Consider the database D and the set of dependencies $\Sigma_{2.6} = \{r\}$ of Example 2.6. Recall that there is no standard Chase step involving D and r. As the core Chase starts by applying all standard Chase steps, the only core Chase sequence is the empty one, similar to the standard Chase case. □

In the following, whenever a successful terminating $\star$-chase sequence of D with Σ does exist, where $\star \in \{\mathsf{std}, \mathsf{obl}, \mathsf{sobl}, \mathsf{core}\}$ stands for the standard, oblivious, semi-oblivious, and core Chase, respectively, we use $chase^\star(D, \Sigma)$ to denote one of the homomorphically equivalent universal models constructed by the $\star$-chase. If there is a failing $\star$-chase sequence of D with Σ, we write $chase^\star(D, \Sigma) = \perp$.

3

Termination of programs with function symbols

As discussed in the Introduction, the use of function symbols in the context of logic programming gives rise to a powerful language, able to encode many interesting problems in an very concise and elegant way, but at the same time, termination of the inference process becomes undecidable. For this reason, the logic programming community has been motivated to identify subclasses of this powerful language for which termination of the programs evaluation is guaranteed. This chapter is devoted to introduce the notion of limited programs, which is a more formal and general definition of terminating program, along with some useful notions and definitions regarding the analysis of limited programs. Then, the best-known decidable classes in the literature of limited programs are introduced along with the contributions of this thesis in the field of logic programs termination.

3.1 Limited programs

Let $\mathcal{P}$ be a program and let $\mathcal{B} \subseteq \mathit{pred}(\mathcal{P})$. An argument $p[i] \in \mathit{args}(\mathcal{P})$ is said to be *limited w.r.t.* $\mathcal{B}$, if for every finite set of facts D for which $\mathit{pred}(D) = \mathcal{B}$ and every model $M \in \mathcal{SM}(\mathcal{P} \cup D)$ the set $\{t_i \mid p(t_1, \ldots, t_n) \in M\}$ is finite. $\mathcal{P}$ is *limited w.r.t.* $\mathcal{B}$ if every argument in $\mathit{args}(\mathcal{P})$ is limited w.r.t. $\mathcal{B}$. Equivalently, if $\mathcal{P}$ is limited w.r.t. $\mathcal{B}$, it means that $\mathcal{P} \cup D$ admits a finite number of stable models and each is of finite size, that is, $|\mathcal{SM}(\mathcal{P} \cup D)|$ is finite and every stable model $M \in \mathcal{SM}(\mathcal{P} \cup D)$ is finite. Finally, given a program $\mathcal{P}$, an argument $p[i] \in \mathit{args}(\mathcal{P})$ (resp. $\mathcal{P}$) is *limited* if $p[i]$ (resp. $\mathcal{P}$) is limited w.r.t. $\mathit{pred}(\mathcal{P})$.

If we focus on positive normal programs, the notion of limitedness can be equivalently defined as follows. A positive normal program $\mathcal{P}$ is *limited* w.r.t.

$\mathcal{B} \subseteq \mathit{pred}(\mathcal{P})$, if for every finite set of facts D for which $\mathit{pred}(D) = \mathcal{B}$, there is a finite natural number n such that $T^n_{\mathcal{P}\cup D}(\varnothing) = T^{\infty}_{\mathcal{P}\cup D}(\varnothing)$. We call such programs *terminating w.r.t.* $\mathcal{B}$. Furthermore, a positive normal program $\mathcal{P}$ is terminating if it is terminating w.r.t. $\mathit{pred}(\mathcal{P})$.

3.2 Analysis of limited programs

It is worth mentioning that checking limitedness of a logic program $\mathcal{P}$ (w.r.t. some $\mathcal{B} \subseteq \mathit{pred}(\mathcal{P})$) could be carried on by checking the termination of a positive normal program $\mathit{st}(\mathcal{P})$ derived from $\mathcal{P}$ as follows. Every rule $A_1 \vee \ldots \vee A_m \leftarrow \mathit{body}$ in $\mathcal{P}$ is replaced with m positive normal rules of the form $A_i \leftarrow \mathit{body}^+$ $(1 \leq i \leq m)$ where body^+ is obtained from body by deleting all negative literals. In fact, the minimal model of $\mathit{st}(\mathcal{P})$ contains every stable model of $\mathcal{P}$—whence, finiteness and computability of the minimal model of $\mathit{st}(\mathcal{P})$ implies that $\mathcal{P}$ has a finite number of stable models, each of finite size, which can be computed [17].

Also, notice that the notion of limited (or terminating) program $\mathcal{P}$ w.r.t. to $\mathcal{B} \subseteq \mathit{pred}(\mathcal{P})$ does not change if we focus only on finite sets of facts D containing only constants, as every fact in D of the form $b(f(c), d)$ can be simulated as follows:

- replace $b(f(c), d)$ in D with the atom $b'(c, d)$, where b' is a fresh new predicate symbol;
- add a rule of the form $b(f(X), Y) \leftarrow b'(X, Y)$ to $\mathcal{P}$;
- replace each occurrence in $\mathcal{B}$ of the predicate symbol b, with b'.

If we denote with $\mathcal{P}'$, D' and $\mathcal{B}'$ the program, set of facts and set of predicate symbols obtained by applying the transformation above to $\mathcal{P}$, D and $\mathcal{B}$, it is easy to see that $\mathcal{P} \cup D$ admits a finite number of stable models and each is of finite size iff $\mathcal{P}' \cup D'$ admits a finite number of stable models and each is of finite size.

Finally, notice also that given program $\mathcal{P}$, if we consider $\mathcal{B} = \mathit{pred}(\mathcal{P})$ (i.e. we consider simply limited programs), $\mathcal{P}$ is limited w.r.t. $\mathcal{B}$ iff the program obtained from $\mathcal{P}$ by deleting all its facts is limited w.r.t. $\mathcal{B}$. In some cases, logic programs will be assumed w.l.o.g. to enjoy some (or all) of the properties above, in order to simply the discussion.

As stated in the previous section, we are interested in finding whether a given logic program admits a finite number of finite stable models, for *every*

set of facts D. This means that whenever we are given a program $\mathcal{P}$, the analysis of its limitedness will be carried on by considering $\mathcal{P}$ alone, without considering a specific set of facts. For this reason, all the complexity results regarding checking whether a program is limited will be given w.r.t. the size of the program itself.

Given a logic program $\mathcal{P}$, we assume that constant space is used to store each constant, logical variable, function symbol, and predicate symbol of $\mathcal{P}$. The *syntactic size*[1] of a term t (resp. (set of) atom, rule, program), denoted by $||t||$, is the number of symbols occurring in t, except for the symbols "(", ")", ",", ".", and "$\leftarrow$". Thus, in the following sections, the complexity of a problem involving $\mathcal{P}$ is assumed to be w.r.t. $||\mathcal{P}||$.

3.3 Termination criteria

In the following, we present some of the most important classes of limited logic programs with function symbols known in the literature.

3.3.1 *Ω*-restriction

ω-restricted programs is the simplest class of limited programs, introduced in [76], where logic programs are admitted to be normal and with negation. The basic idea is to identify a stratification of predicates for a given program. This stratification generalizes classical stratification by introducing an additional stratum, called ω-stratum, that holds all the unstratifiable predicates, that is, all predicate symbols depending negatively on each other.

First, we need to define a particular graph used by the techinque, that we call ω-graph.

Definition 3.1. *The* ω-graph $G^\omega(\mathcal{P})$ *of a program* $\mathcal{P}$ *is a labelled directed graph whose nodes are the predicate symbols appearing in* $\mathcal{P}$ *and there is a* positive *edge* $(q, p, +)$ *(resp.* negative *edge* $(q, p, -)$*) from* q *to* p *iff there is a rule* r *of* $\mathcal{P}$ *with* p *appearing in* $head(r)$ *and* q *appearing in* $body^+(r)$ *(resp.* $body^-(r)$*). We say that a path on* $G^\omega(\mathcal{P})$ *is* positive *if all its edges are labelled with "+",* negative *otherwise.* □

[1] We use the name syntactic size to distinguish it from the notion of size introduced in Definition 3.73.

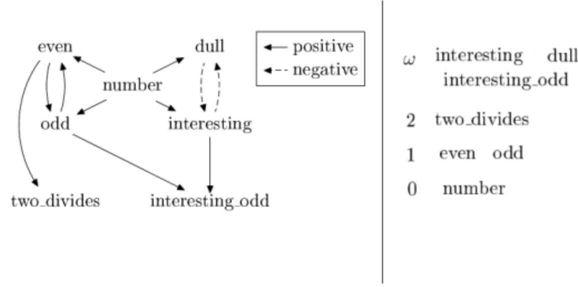


Fig. 3.1: ω-graph of $\mathcal{P}_{3.3}$ (left) and ω-stratification S of $\mathcal{P}$ (right).

Now we define the notion of ω-stratification, which generalizes classic stratification by adding a new stratum, called ω-stratum.

Definition 3.2. *An* ω-stratification *of a program* $\mathcal{P}$ *is a function* $S : pred(\mathcal{P}) \rightarrow \mathbb{N} \cup \{\omega\}$*, where by convention* $\forall n \in \mathbb{N}\ \omega > n$*, such that for every path in* $G^{\omega}(\mathcal{P})$ *from* q *to* p*,* i) $S(p) \geq S(q)$ *if the path is positive, and* ii) $S(p) > S(q) \vee S(p) = \omega$ *otherwise.* □

Example 3.3. Consider the following program $\mathcal{P}_{3.3}$:

$$
\begin{array}{ll}
\quad\ \mathtt{number(0).} & \ldots\ \mathtt{number(n).} \\
\quad\ \mathtt{even(0).} & \\
\mathtt{r_1 : odd(X+1)} & \leftarrow \mathtt{number(X), even(X).} \\
\mathtt{r_2 : even(X+1)} & \leftarrow \mathtt{number(X), odd(X).} \\
\mathtt{r_3 : two_divides(X)} & \leftarrow \mathtt{even(X).} \\
\mathtt{r_4 : interesting(X)} & \leftarrow \mathtt{number(X), \neg dull(X).} \\
\mathtt{r_5 : dull(X)} & \leftarrow \mathtt{number(X), \neg interesting(X).} \\
\mathtt{r_6 : interesting_odd(X)} & \leftarrow \mathtt{odd(X), interesting(X).}
\end{array}
$$

Where $X+1$ is a shortand for the function symbol $+1$ applied to X, i.e. $+1(X)$. The ω-graph is shown in Figure 3.1. A possible ω-stratification shown in Figure 3.1 is as follows: $S(\mathtt{number}) = 0$, $S(\mathtt{even}) = S(\mathtt{odd}) = 1$, $S(\mathtt{two_divides}) = 2$, $S(\mathtt{interesting}) = S(\mathtt{dull}) = S(\mathtt{interesting_odd}) = \omega$. □

It is simple to note that every normal program always admits an ω-stratification, i.e. the stratification placing every predicate in the ω-stratum. Now we can present the definition of ω-restricted programs.

Definition 3.4. *The* ω-valuation of a rule $r : p(\boldsymbol{t}) \leftarrow body(r)$ *under an* ω*-stratification* S *is the function* $\Omega(r, S) = S(p)$*. The* ω-valuation of a vari-

able X *in a rule* r *under an* ω*-stratification* S *is the function* $\Omega(X, r, S) = min(\{S(q) \mid q(\boldsymbol{u}) \in body^+(r) \wedge X \text{ occurs in } \boldsymbol{u}\} \cup \{\omega\})$. □

Definition 3.5. *A program* $\mathcal{P}$ *is* ω-restricted *iff every rule in* $\mathcal{P}$ *is* ω*-restricted. A rule* r *of a program* $\mathcal{P}$ *is* ω*-restricted iff there exists an* ω*-stratification* S *of* $\mathcal{P}$ *s.t. for every variable* X *occurring in* r*, the condition* $\Omega(X, r, S) < \Omega(r, S)$ *holds. The class of* ω*-restricted programs is denoted by* ΩR □

Example 3.6. Consider the program $\mathcal{P}_{3.3}$ and the ω-stratification S defined in Example 3.3. The ω-valuation of rule r_4 is $\Omega(r_4, S) = S(\texttt{interesting}) = \omega$. The ω-valuation of variable X in r_4 is $\Omega(X, r_4, S) = min(S(\texttt{number}), \omega) = min(0, \omega) = 0$. Rule r_4 is ω-restricted, since $\Omega(X, r_4, S) < \Omega(r_4, S)$, for the unique variable X occurring in r_4. Moreover, every rule in $\mathcal{P}_{3.3}$ is ω-restricted, thus $\mathcal{P}_{3.3}$ is ω-restricted. □

3.3.2 λ-restriction

The λ-restricted technique has been introduced by Gebser et al. in [44] as a syntactic tool of the GrinGo system for checking the existence of a finite ground instantiation of a given logic program. In the same way as ω-restricted, the λ-restricted criterion has been defined over normal logic programs admitting negation.

The definition of λ-restricted program is very simple and can be stated as follows.

Definition 3.7. *A program* $\mathcal{P}$ *is* λ-restricted *if there is a function (level mapping)* $\lambda : pred(\mathcal{P}) \to \mathbb{N}$ *s.t. for every predicate symbol* p *in* $\mathcal{P}$*:*

$$\max\{\max\{\min\{\lambda(q) \mid q(\boldsymbol{u}) \in body^+(r) \ \wedge \ X \text{ occurs in } \boldsymbol{u}\} \mid X \text{ occurs in } r\} \mid r \text{ defines } p\} < \lambda(p)$$

The class of λ*-restricted programs is denoted by* λR. □

Intuitively, this criterion tries to verify whether a given predicate p is "bounded" in every rule which defines it by predicates in the body. Moreover, the level mapping λ does not define a stratification of predicates like the ω-restricted approach.

Example 3.8. Consider the following program $\mathcal{P}_{3.8}$:

```
r0 : a(1).
r1 : b(X) ← a(X), c(X).
r2 : c(X) ← a(X).
r3 : c(X) ← b(X).
```

It easy to see that there is a level mapping λ defined as $\lambda(a) = 0$, $\lambda(b) = 1$ and $\lambda(c) = 2$ which satisfies Definition 3.7. □

It has also been shown that λ-restricted programs strictly contain the ω-restricted ones.

Proposition 3.9. *[44]* $\Omega R \subset \lambda R$ □

3.3.3 Finite-domain

The finite-domain technique was introduced in [26]. It was the first approach working at the argument level, i.e. the positions inside predicates of the given program. The previous approaches are defined to find a level mapping for *predicates*. The finite-domain technique instead, tries to identify limited arguments. If all the arguments are found to be limited, then the program is guaranteed to be limited. This technique is defined over normal programs with negation.

Definition 3.10. *The* argument graph *of a program $\mathcal{P}$, denoted $G^{Arg}(\mathcal{P})$, is a directed graph such that the set of nodes of $G^{Arg}(\mathcal{P})$ is args$(\mathcal{P})$ and there is an edge from $q[j]$ to $p[i]$, denoted by $(q[j], p[i])$, iff there is a rule $r \in \mathcal{P}$ such that* i) *an atom $p(t_1, ..., t_n)$ appears in head(r),* ii) *an atom $q(u_1, ..., u_m)$ appears in body$^+(r)$ and* iii) *terms t_i and u_j have a common variable.* □

We now show the formal definition of finite-domain programs. We say that two arguments $p[i], q[j]$ are recursive in $G^{Arg}(\mathcal{P})$ if there is a path from $p[i]$ to $q[j]$ and viceversa.

Definition 3.11. *The set of* finite-domain arguments *($\mathcal{FD}$ arguments) of a program $\mathcal{P}$ is the maximal set (w.r.t. set inclusion) $FD(\mathcal{P})$ of arguments $q[k]$ of $\mathcal{P}$ s.t., for every rule r with $q[k]$ occurring in head(r), the term t of $q[k]$ in r is such that:*

1. *t is a ground term, or*
2. *t is a subterm of a term occurring in a $\mathcal{FD}$ argument $p[i]$ of body$^+(r)$, or*

3. *every variable occurring in t also occurs in the term of a $\mathcal{FD}$ argument $p[i]$ of $body^+(r)$ s.t. $p[i]$ and $q[k]$ are recursive in $G^{Arg}(\mathcal{P})$.*

$\mathcal{P}$ *is* finite-domain *iff all the arguments of* $\mathcal{P}$ *are finite-domain. The class of finite-domain programs is denoted by* FD □

Example 3.12. Consider the following program $\mathcal{P}_{3.12}$:

$$
\begin{aligned}
&\mathtt{r}_0 : \mathtt{s(1)}.\\
&\mathtt{r}_1 : \mathtt{p(X)} \quad\;\; \leftarrow \mathtt{p(f(X))}.\\
&\mathtt{r}_2 : \mathtt{p(f(X))} \leftarrow \mathtt{p(X)}, \mathtt{s(X)}.
\end{aligned}
$$

It easy to see that this program is finite-domain. Computing the set of finite-domain arguments can be done as follows: initially assume that $FD(\mathcal{P}) = arg(\mathcal{P})$, then remove iteratively from $FD(\mathcal{P})$ every argument not satisfying Definition 3.11. In this case, $FD(\mathcal{P}) = \{\mathtt{s[1]}, \mathtt{p[1]}\}$. Obviously for $s[1]$, rule r_0 satisfies condition 1 of Definition 3.11; for $p[1]$ rule r_1 satisfies condition 2 and rule r_2 satisfies condition 3, since the unique variable X occurring in $f(X)$ occurs in $s(X)$, $s[1]$ is finite-domain and it is not recursive with $p[1]$. □

Intuitively, every condition in the definition of finite-domain arguments deals with a different type of rule in the input program. The first one identifies $q[k]$ as possibly finite-domain if it is bounded directly in the program by facts or rules with ground terms in their head that define it. The second condition guarantees that the term associated to an $\mathcal{FD}$ argument $p[i]$ in the positive body is an "upper bound" for the term of $q[k]$. Finally, condition 3 is applied to rules that could "grow" the term associated to the argument $q[k]$. To bound such a term, the positive body argument $p[i]$ must be finite-domain and not recursive with $q[k]$. Note that both conditions of non recursivity and $p[i] \in \mathcal{FD}(\mathcal{P})$ are important, since the former prevents the actual rule to produce infinite terms and the latter prevents that even when no recursion is found between $q[k]$ and $p[i]$, $q[k]$ is not recursive with some other $\mathcal{FD}$ argument, unless $p[i]$ satisfies condition 2.

It has been shown that finite-domain programs strictly contain ω-restricted programs but are incomparable with λ-restricted ones.

Proposition 3.13. *[26]*

1. $\Omega R \subset FD$;
2. FD *and* λR *are incomparable.* □

3.3.4 Argument ranking

The argument ranking of a program has been proposed in [60] to define the class AR of *argument-restricted* programs. Argument-restricted programs allow the presence of disjunction in the head and negation in rules.

An *argument ranking* for a program $\mathcal{P}$ is a partial function ϕ from $args(\mathcal{P})$ to non-negative integers, called *ranks*, such that, for every rule r of $\mathcal{P}$, every atom $p(t_1, ..., t_n)$ occurring in the head of r, and every variable X occurring in a term t_i, if $\phi(p[i])$ is defined, then $body^+(r)$ contains an atom $q(u_1, ..., u_m)$ such that X occurs in a term u_j, $\phi(q[j])$ is defined, and the following condition is satisfied

$$\phi(p[i]) - \phi(q[j]) \geq d(X, t_i) - d(X, u_j). \tag{3.1}$$

A program $\mathcal{P}$ is said to be *argument-restricted* if it has an argument ranking assigning ranks to all arguments of $\mathcal{P}$.

Example 3.14. Consider the following program $P_{3.14}$:

$$r_1 : \mathtt{p(f(X))} \leftarrow \mathtt{p(X), b(X)}.$$
$$r_2 : \mathtt{t(f(X))} \leftarrow \mathtt{p(X)}.$$
$$r_3 : \mathtt{s(X)} \leftarrow \mathtt{t(f(X))}.$$

This program has an argument ranking ϕ, where $\phi(\mathtt{b[1]})= 0$, $\phi(\mathtt{p[1]})= 1$, $\phi(\mathtt{t[1]})= 2$ and $\phi(\mathtt{s[1]})= 1$. Consequently, $P_{3.14}$ is argument-restricted. □

Intuitively, the rank of an argument is an estimation of the depth of terms that may occur in it. In particular, let d_1 be the rank assigned to a given argument $p[i]$ and let d_2 be the maximal depth of terms occurring in the facts. Then $d_1 + d_2$ gives an upper bound of the depth of terms that may occur in $p[i]$ during the program evaluation. Different argument rankings may satisfy condition (3.1). A function assigning minimum ranks to arguments is denoted by ϕ_{min}.

Minimum ranking. We define a monotone operator Ω that takes as input a function ϕ over arguments and gives as output a function over arguments that gives an upper bound of the depth of terms.
More specifically, we define $\Omega(\phi)(p[i])$ as

$$max(max\{D(p(t_1, ..., t_n), r, i, X) \mid r \in \mathcal{P} \wedge p(t_1, ..., t_n) \in head(r) \wedge X \, occurs \, in \, t_i\}, 0)$$

where $D(p(t_1, ..., t_n), r, i, X)$ is defined as

$min\{d(X,t_i) - d(X,u_j) + \phi(q[j]) \mid q(u_1,...,u_m) \in body^+(r) \wedge X\, occurs\, in\, u_j\}$.

In order to compute ϕ_{min} we compute the fixpoint of Ω starting from the function ϕ_0 that assigns 0 to all arguments. In particular, we have:

$$\phi_0(p[i]) = 0;$$
$$\phi_k(p[i]) = \Omega(\phi_{k-1})(p[i]) = \Omega^k(\phi_0)(p[i]).$$

The function ϕ_{min} is defined as follows:

$$\phi_{min}(p[i]) = \begin{cases} \Omega^k(\phi_0)(p[i]) & if\ \exists k \text{ (finite) s.t. } \Omega^k(\phi_0)(p[i]) = \Omega^\infty(\phi_0)(p[i]) \\ undefined & otherwise \end{cases}$$

We denote the set of *restricted arguments* of $\mathcal{P}$ as $AR(\mathcal{P}) = \{p[i] \mid p[i] \in args(\mathcal{P}) \wedge\ \phi_{min}(p[i])$ is defined$\}$. Clearly, from definition of ϕ_{min}, it follows that all restricted arguments are limited. $\mathcal{P}$ is said to be *argument-restricted* iff $AR(\mathcal{P}) = args(\mathcal{P})$. The class of argument-restricted programs is denoted by AR.

Example 3.15. Consider again program $P_{3.14}$ from Example 3.14. The following table shows the first four iterations of Ω starting from the base ranking function ϕ_0:

	ϕ_0	$\phi_1 = \Omega(\phi_0)$	$\phi_2 = \Omega(\phi_1)$	$\phi_3 = \Omega(\phi_2)$	$\phi_4 = \Omega(\phi_3)$
b[1]	0	0	0	0	0
p[1]	0	1	1	1	1
t[1]	0	1	2	2	2
s[1]	0	0	0	1	1

Since $\Omega(\phi_3) = \Omega(\phi_2)$, further applications of Ω provide the same result. Consequently, ϕ_{min} coincides with ϕ_3 and defines ranks for all arguments of $P_{3.14}$. □

Let $M = |args(\mathcal{P})| \times d_{max}$, where d_{max} is the largest depth of terms occurring in the heads of rules of $\mathcal{P}$. One can determine whether $\mathcal{P}$ is argument-restricted by iterating Ω starting from ϕ_0 until

- one of the values of $\Omega^k(\phi_0)$ exceeds M, in such a case $\mathcal{P}$ is not argument-restricted;

- $\Omega^{k+1}(\phi_0) = \Omega^k(\phi_0)$, in such a case ϕ_{min} coincides with ϕ_k, ϕ_{min} is total, and $\mathcal{P}$ is argument-restricted.

Observe that if the program is not argument-restricted the first condition is verified with $k \leq M \times |args(\mathcal{P})| \leq M^2$, as at each iteration the value assigned to at least one argument is changed. It has been shown that the class of argument-restricted programs strictly includes ω-restricted, λ-restricted and finite-domain programs.

Proposition 3.16. *[60] $\mathcal{T} \subset AR$, for $\mathcal{T} \in \{\Omega R, \lambda R, FD\}$.* □

Unfortunately, no result was established regarding the complexity of computing the set of restricted-arguments of a program.

As we will show in Section 3.5, computing the set of restricted-arguments of a program $\mathcal{P}$ is feasible in $PTime$.

3.4 Argument-based approach: Mapping-restriction

In this section we present one of the most general techniques aimed to identify a logic program with function symbols as terminating. In particular, it has been shown to be the most general criterion based on the sole analysis of the arguments of a given logic program [18]. We start by introducing notations and terminology used hereafter.

Let $\mathcal{P}$ be a program. We define $\bar{\mathcal{B}} = \mathit{pred}(\mathcal{P}) - \mathit{def}(\mathcal{P})$, that is the set of predicates symbols occurring in $\mathcal{P}$ but not occurring in the head of some rule of $\mathcal{P}$. We will focus on the termination of logic programs $\mathcal{P}$ w.r.t. $\bar{\mathcal{B}}$, thus sets of facts are understood to contain only facts whose predicate symbols occur in $\bar{\mathcal{B}}$. Terminating programs of this form are of particular interest, as they are able to encode database-related problems, where predicates in $\bar{\mathcal{B}}$ can be used to define database only facts. Furthermore, due to the presence of function symbols, as we will show later in this chapter, they are also able model temporal phenomena.

We assume that the termination analysis is restricted to sets of facts D and programs $\mathcal{P}$ such that:

- D contains only constants;
- the depth of complex terms occurring in $\mathcal{P}$ is at most one;
- No constants appear in $\mathcal{P}$;

There is no real restriction in such assumptions as every program $\mathcal{P} \cup D$ could be rewritten into an equivalent program satisfying such conditions. The first condition is w.l.o.g. as shown in Section 3.1. The second condition can be satisfied by rewriting rules of $\mathcal{P}$ of the form $p(f(h(X))) \leftarrow q(X)$ into two rules: $p(f(X)) \leftarrow p'(X)$, $p'(h(X)) \leftarrow q(X)$. For the third condition, rules of $\mathcal{P}$ of the form $p(a) \leftarrow body(X, a)$, where a is a constant, could be rewritten as $p(Y) \leftarrow body(X, Y), p'(Y)$ with the addition of $p'(a)$ to D.

Thus, in the rest of this section, we assume that every program $\mathcal{P}$ and set of facts D satisfy the conditions above.

Definition 3.17. Given a program $\mathcal{P}$, an *m-set* $U_{\mathcal{P}}$ is a set of pairs $p[i]/s$, called *mappings*, such that $p[i] \in args(\mathcal{P})$ and $s \in F^*_{\mathcal{P}}$. □

Intuitively, a pair $p[i]/s$ means that during the evaluation of the program, considering all possible sets of facts, the argument $p[i]$ could take values whose structure, in terms of nesting of function symbols, is described by s. For instance, let $p(f(g(c_1)), c_2)$ be a ground atom derivable through the evaluation of the input program, the mappings for its arguments are $p[1]/fg$ and $p[2]/\epsilon$.

Definition 3.18. *Let $\mathcal{P}$ be a program, D a finite set of facts instance and M a model of $\mathcal{P} \cup D$. We denote by U_M the* m-set derivable from M *defined as follows:*

$$U_M = \{p[i]/s \mid p(t_1, ..., t_n) \in M \ \wedge \ s \in strings(t_i)\}$$

where, $strings(t)$ denotes the set of strings recursively defined as:

1. *$strings(t) = \{\epsilon\}$ if t is a constant, or*
2. *$strings(t) = \{f \cdot s \mid t = f(u_1, ..., u_k) \ \wedge \ s \in strings(u_j) \ \wedge \ 1 \leq j \leq k\}$ otherwise.* □

Given an m-set $U_{\mathcal{P}}$ and an atom $p(t_1, ..., t_n)$ occurring in $\mathcal{P}$, we say that an occurrence of a variable X in t_i has a *mapping* to a string s in $U_{\mathcal{P}}$ if $p[i]/s \in U_{\mathcal{P}} \wedge t_i = X$ or $p[i]/gs \in U_{\mathcal{P}} \wedge t_i = g(...X...)$. For instance, considering an atom $p(f(X))$ and $U_{\mathcal{P}} = \{p[1]/fg\}$, the occurrence of X in $f(X)$ has a mapping to the string g in $U_{\mathcal{P}}$.

Definition 3.19. *Let $\mathcal{P}$ be a program and let $U_{\mathcal{P}}$ be an m-set. We say that $U_{\mathcal{P}}$ is a* supported *m-set of $\mathcal{P}$ if:*

1. *$q[j]/\epsilon \in U_{\mathcal{P}}$ for every argument $q[j] \in args(\bar{\mathcal{B}})$, and*

2. *for every rule $r \in \mathcal{P}$ and for every variable X in r, if all occurrences of variable X in the body of r have a mapping to a string s in $U_\mathcal{P}$, then all occurrences of X in the head of r also have a mapping to s in $U_\mathcal{P}$.* □

Intuitively, if $U_\mathcal{P}$ is a supported m-set of $\mathcal{P}$, for every finite set of facts D there exists a model M of $\mathcal{P} \cup D$ such that $U_M \subseteq U_\mathcal{P}$. The number of supported m-sets for a given program $\mathcal{P}$ could be infinite, and there can be supported m-sets of infinite size.

Given a program $\mathcal{P}$, a supported m-set $U_\mathcal{P}$ of $\mathcal{P}$ is *minimal* if there is no supported m-set $U'_\mathcal{P}$ of $\mathcal{P}$ such that $U'_\mathcal{P} \subset U_\mathcal{P}$. It is simple to note that every program $\mathcal{P}$ has a unique supported minimal m-set, called *minimum supported m-set*, denoted in the following by $U^*_\mathcal{P}$. The minimum supported m-set can be obtained as the intersection of all supported m-sets of $\mathcal{P}$.

Example 3.20. Consider the following program $\mathcal{P}_{3.20}$

$$\begin{aligned}
r_1 &: \mathtt{p(X,f(X))} && \leftarrow \mathtt{b(X)}.\\
r_2 &: \mathtt{p(f(X),X)} && \leftarrow \mathtt{b(X)}.\\
r_3 &: \mathtt{q(f(X),g(X))} && \leftarrow \mathtt{p(X,X)}.\\
r_4 &: \mathtt{q(f(X),f(X))} && \leftarrow \mathtt{q(X,X)}.
\end{aligned}$$

where $\bar{\mathcal{B}} = pred(\mathcal{P}) - def(\mathcal{P}) = \{\mathtt{b}\}$. If $D = \{\mathtt{b(a)}\}$, the minimum model of $\mathcal{P}_{3.20} \cup D$ and the corresponding m-set are

$$\begin{aligned}
\mathcal{MM}(\mathcal{P}_{3.20} \cup D) &= \{\mathtt{b(a)}, \mathtt{p(a,f(a))}, \mathtt{p(f(a),a)}\},\\
U_{\mathcal{MM}(\mathcal{P}_{3.20} \cup D)} &= \{\mathtt{b}[1]/\epsilon, \mathtt{p}[1]/\epsilon, \mathtt{p}[2]/\mathtt{f}, \mathtt{p}[1]/\mathtt{f}, \mathtt{p}[2]/\epsilon\}.
\end{aligned}$$

The minimum supported m-set of this program is

$$U^*_{\mathcal{P}_{3.20}} = \{\mathtt{b}[1]/\epsilon, \mathtt{p}[1]/\epsilon, \mathtt{p}[2]/\mathtt{f}, \mathtt{p}[1]/\mathtt{f}, \mathtt{p}[2]/\epsilon, \mathtt{q}[1]/\mathtt{f}, \mathtt{q}[2]/\mathtt{g}, \mathtt{q}[1]/\mathtt{ff}, \mathtt{q}[2]/\mathtt{gf}\},$$

that is a finite proper superset of $U_{\mathcal{MM}(\mathcal{P}_{3.20} \cup D)}$. □

Definition 3.21. *Given a program $\mathcal{P}$, an argument $p[i] \in arg(\mathcal{P})$ is* mapping-restricted *(*m-restricted *for short) iff $U^*_\mathcal{P}$ contains a finite set (possibly empty) of mappings $p[i]/s$. $MR(\mathcal{P})$ denotes the set of all* m-restricted arguments *of $\mathcal{P}$. A program $\mathcal{P}$ is* m-restricted *if $MR(\mathcal{P}) = arg(\mathcal{P})$, i.e. it admits a finite supported m-set. The set of m-restricted programs is denoted by $\mathcal{MR}$.* □

From the discussion above it follows that each program whose minimum supported m-set is finite, has a finite minimum model for every finite set of

facts D. Moreover, it can be shown that every m-restricted argument is limited w.r.t. $\bar{\mathcal{B}}$.

Theorem 3.22. *Every program $\mathcal{P}$ admitting a finite supported m-set is terminating w.r.t. $\bar{\mathcal{B}}$.*

Proof. To prove the theorem, first we show that, given the particular set of facts $D^\epsilon = \{b(\epsilon, ..., \epsilon) \mid b \in \bar{\mathcal{B}}\}$, $U_{\mathcal{MM}(\mathcal{P} \cup D)} \subseteq U_{\mathcal{MM}(\mathcal{P} \cup D^\epsilon)}$, for every set of facts D.

Let $M^* = \mathcal{MM}(\mathcal{P} \cup D)$ and $M_\epsilon = \mathcal{MM}(\mathcal{P} \cup D^\epsilon)$, we have that:

1. given an argument $b[i] \in args(\bar{\mathcal{B}})$, if $b[i]/\epsilon \in U_{M^*}$, then $b[i]/\epsilon \in U_{M_\epsilon}$ by definition of D^ϵ;
2. given an argument $p[i] \notin args(\bar{\mathcal{B}})$, if $p[i]/s \in U_{M^*}$, then there must be a rule $r : p(t_1, ..., t_i, ..., t_n) \leftarrow body(r)$ in $\mathcal{P}$ and a substitution θ for all variables occurring in r, such that all atoms in $body(r)\theta$ appear in M^* (that is, the atom is derived using the immediate consequence operator $T_\mathcal{P}$). Let θ' be the substitution obtained from θ by replacing all constants occurring in θ with ϵ. Since $\mathcal{P}$ is a *positive* program, θ' is such that atoms in $body(r)\theta'$ and $head(r)\theta'$ occur in M_ϵ. Then, by construction of θ', we have that if $p[i]/s \in U_{M^*}$, then $p[i]/s \in U_{M_\epsilon}$.

Now, we show that $U_{M_\epsilon} \subseteq U^*_\mathcal{P}$.

1. given an argument $b[i] \in args(\bar{\mathcal{B}})$, if $b[i]/\epsilon \in U_{M_\epsilon}$, then $b[i]/\epsilon \in U^*_\mathcal{P}$ by definition of M_ϵ and $U^*_\mathcal{P}$;
2. given an argument $p[i] \notin args(\bar{\mathcal{B}})$, if $p[i]/s \in U_{M_\epsilon}$, then there must be a rule $r : p(t_1, ..., t_i, ..., t_n) \leftarrow body(r)$ in $\mathcal{P}$ and a substitution θ for all variables occurring in r, such that all atoms in $body(r)\theta$ appear in M_ϵ. Now, consider the substitution θ' obtained from θ taking only the pairs $X/t \in \theta$ such that X appears in t_i. It is easy to see that this substitution represents the fact that all the occurrences of X in $body(r)$ have a mapping to s in $U^*_\mathcal{P}$, and then $p[i]/s \in U^*_\mathcal{P}$.

Finally, since for every set of facts D, $U_{M^*} \subseteq U_{M_\epsilon}$ and $U_{M_\epsilon} \subseteq U^*_\mathcal{P}$, it follows that $U_{M^*} \subseteq U^*_\mathcal{P}$ for every set of facts D. Then, if $U^*_\mathcal{P}$ is finite, $M^* = \mathcal{MM}(\mathcal{P} \cup D)$ is finite. □

Proposition 3.23. *Given a program $\mathcal{P}$, every m-restricted argument is limited w.r.t. $\bar{\mathcal{B}}$.*

Proof. Let $p[i]$ be a m-restricted argument of $\mathcal{P}$. By definition of m-restricted argument, $U^*_{\mathcal{P}}$ contains a finite set of mappings $p[i]/s$. Since, for every set of facts D, $U_{\mathcal{MM}(\mathcal{P}\cup D)} \subseteq U^*_{\mathcal{P}}$, from the proof of Theorem 3.22, $U_{\mathcal{MM}(\mathcal{P}\cup D)}$ contains a finite set of mappings $p[i]/s$, then, the set $\{t_i \mid p(t_1, ..., t_i, ..., t_n) \in \mathcal{MM}(\mathcal{P} \cup D)\}$ is finite, thus $p[i]$ is limited w.r.t. $\bar{\mathcal{B}}$. □

Detecting m-restricted arguments. In order to construct the set of m-restricted arguments of a given program $\mathcal{P}$, we introduce a program $\mathcal{P}^u \cup D^u$, obtained as a transformation of $\mathcal{P}$ and having the following properties:

- checking whether $\mathcal{MM}(\mathcal{P}^u \cup D^u)$ is finite is decidable;
- $U^*_{\mathcal{P}^u \cup D^u} = U_{M^*}$, where $M^* = \mathcal{MM}(\mathcal{P}^u \cup D^u)$;
- There is a bijection h from $args(\mathcal{P})$ to $args(\mathcal{P}^u \cup D^u)$ s.t. $h(U^*_{\mathcal{P}}) = U^*_{\mathcal{P}^u \cup D^u}$, i.e. $p[i]/s \in U^*_{\mathcal{P}}$ iff $h(p[i])/s \in U^*_{\mathcal{P}^u \cup D^u}$.

Definition 3.24. *Let $\mathcal{P}$ be a program. Then:*

- *D^u is the set of facts defined as $\{b_i(0) \mid b[i] \in args(\bar{\mathcal{B}})\}$;*
- *$\mathcal{P}^u$ denotes the program derived from $\mathcal{P}$ as follows:*
  - *for every rule $r = p(t_1, ..., t_n) \leftarrow body$ in $\mathcal{P}$, for every variable X occurring in $p(t_1, ..., t_n)$, and for every term t_i where X occurs, $\mathcal{P}^u$ contains a rule:*

$$p_i(t_i^X) \leftarrow \bigwedge_{\substack{q(u_1,...,u_k) \text{ in } body \\ \wedge\ X \text{ occurs in } u_j}} q_j(u_j^X)$$

where t^X is defined as follows:

$$t^X = \begin{cases} X & \text{if } t = X \\ f(X) & \text{if } t = f(..., X, ...). \end{cases}$$

□

Example 3.25. Consider the following program $\mathcal{P}_{3.25}$:

$$\begin{array}{ll} \mathtt{p(X,X)} & \leftarrow \mathtt{b(X)}. \\ \mathtt{q(f(X),f(X))} & \leftarrow \mathtt{p(X,X)}. \\ \mathtt{p(f(X),X)} & \leftarrow \mathtt{q(X,X)}. \end{array}$$

The minimum supported m-set of this program is

$$U^*_{\mathcal{P}_{3.25}} = \{\mathtt{b}[1]/\epsilon, \mathtt{p}[1]/\epsilon, \mathtt{p}[2]/\epsilon, \mathtt{q}[1]/\mathtt{f}, \mathtt{q}[2]/\mathtt{f}, \mathtt{p}[1]/\mathtt{ff},\ \mathtt{p}[2]/\mathtt{f}\}.$$

The transformed unary program $\mathcal{P}^u_{3.25}$ is:

$$
\begin{array}{ll}
\mathtt{p_1(X)} & \leftarrow \mathtt{b_1(X)}. \\
\mathtt{p_2(X)} & \leftarrow \mathtt{b_1(X)}. \\
\mathtt{q_1(f(X))} & \leftarrow \mathtt{p_1(X), p_2(X)}. \\
\mathtt{q_2(f(X))} & \leftarrow \mathtt{p_1(X), p_2(X)}. \\
\mathtt{p_1(f(X))} & \leftarrow \mathtt{q_1(X), q_2(X)}. \\
\mathtt{p_2(X)} & \leftarrow \mathtt{q_1(X), q_2(X)}.
\end{array}
$$

whereas $D^u_{3.25} = \{\mathtt{b_1(0)}\}$.

Furthermore, we have that $\mathcal{MM}(\mathcal{P}^u_{3.25} \cup D^u_{3.25}) = \{\mathtt{b_1(0), p_1(0), p_2(0), q_1(f(0)), q_2(f(0)), p_1(f(f(0)))}, p_2(f(0))\}$, and $U_{M^*} = \{\mathtt{b_1[1]}/\epsilon, \mathtt{p_1[1]}/\epsilon, \mathtt{p_2[1]}/\epsilon, \mathtt{q_1[1]/f}, \mathtt{q_2[1]/f}, \mathtt{p_1[1]/ff}, \mathtt{p_2[1]/f}\}$. It is easy to see that $U_{M^*} = U^*_{\mathcal{P}^u_{3.25}}$. □

The following proposition states that for every program $\mathcal{P}$, the m-sets of $\mathcal{P}^u$ and $\mathcal{P}$ coincide (up to a bijection h) and are derivable from the minimum model of $\mathcal{P}^u \cup D^u$.

Proposition 3.26. *Let $\mathcal{P}$ be a program and $M^* = \mathcal{MM}(\mathcal{P}^u \cup D^u)$, then $U^*_{\mathcal{P}^u \cup D^u} = U_{M^*}$ and there is a bijection h s.t. $h(U^*_{\mathcal{P}}) = U^*_{\mathcal{P}^u \cup D^u}$.*

Proof. The relation $U^*_{\mathcal{P}^u \cup D^u} = U_{M^*}$ is straightforward from the construction of $\mathcal{P}^u \cup D^u$. The existence of h follows from Definition 3.19 of supported m-set and the construction of $\mathcal{P}^u \cup D^u$: h is defined as $h(p[i]) = p_i[1]$ for every $p[i] \in arg(\mathcal{P})$. □

Let us now compare the presented technique with the most general argument-based approach: argument-restricted technique. Argument-restricted generalizes ω-restricted, λ-restricted and finite domain.

Intuitively, the argument-restricted (AR) criterion derives the set of restricted arguments estimating the depth of complex terms that can be associated with an argument during the evaluation. In particular, it considers the depth of terms in the body and in the head of rules, but it does not test the real possibility to activate a rule starting from a fact in the given set D and does not distinguish different function symbols. The new MR technique overcomes these limitations by introducing the concept of supported m-set, which allows us to describe the form of argument values that are derivable during the evaluation of the program, starting from any set of facts instance and use this information to simulate the evaluation process. Furthermore, to

compute strings associated with head arguments, the current technique also checks that rules can be effectively activated. The following theorem states that the class of m-restricted programs generalizes the class of argument restricted programs.

Theorem 3.27. $\mathcal{AR} \subsetneq \mathcal{MR}$.

Proof. Let $\mathcal{P}$ be an argument-restricted program, we denote by $\mathcal{P}_f$ the logic program obtained from $\mathcal{P}$ by replacing every function symbol occurring in $\mathcal{P}$ with the symbol f, admitting that a function symbol does not have fixed arity. Note that $\mathcal{P}$ is argument restricted iff $\mathcal{P}_f$ is argument restricted. Let ϕ be the minimum argument ranking (which is a total function) of both $\mathcal{P}$ and $\mathcal{P}_f$. We denote by s^k the string of length k of the form $s^k = fs^{k-1}$, where $s^0 = \epsilon$. Let $U_{\mathcal{P}_f} = \{p[i]/s^k \mid p[i] \in \mathit{args}(\mathcal{P}) \wedge 0 \leq k \leq \phi(p[i])\}$. Note that such an m-set is a finite supported m-set for $\mathcal{P}_f$. Assume that $\exists p[i]/s \in U^*_{\mathcal{P}}$ such that $|s| > \phi(p[i])$, then, any supported m-set of $\mathcal{P}_f$ would contain a pair $p[i]/s'$ such that $|s'| > \phi(p[i])$, which contradicts the existence of $U_{\mathcal{P}_f}$. Then, $U^*_{\mathcal{P}}$ is finite and $\mathcal{P}$ is in $\mathcal{MR}$. In order to prove the strict inclusion, observe that program $P_{3.20}$ from Example 3.20 is in $\mathcal{MR}$ but not in AR. □

The inclusion is proper even if the program contains only one function symbol. For instance, program $\mathcal{P}_{3.25}$ from Example 3.25 is in $\mathcal{MR}$ but not in AR.

It is worth noting that although both AR and MR techniques are used to identify decidable subclasses of limited programs, they can also be used to detect, for a given program $\mathcal{P}$, subsets of limited arguments of $\mathcal{P}$. The following proposition states that, given a program $\mathcal{P}$, the set $AR(\mathcal{P})$ of restricted arguments of $\mathcal{P}$ is a subset of the set of $MR(\mathcal{P})$ of m-restricted arguments of $\mathcal{P}$.

Corollary 3.28. *For any program $\mathcal{P}$, $AR(\mathcal{P}) \subseteq MR(\mathcal{P})$.*

Proof. Straightforward from the proof of Theorem 3.27. □

Detecting subsets of limited arguments is relevant even when the input program is not recognized as terminating by a given criterion, as in such cases it is possible to combine different techniques to detect the finiteness of the minimum model.

3.4.1 Complexity

In this section we will study the computational complexity of the problem of computing the set $MR(\mathcal{P})$ of m-restricted arguments for a given program $\mathcal{P}$. When $MR(\mathcal{P})$ coincides with $args(\mathcal{P})$, the program $\mathcal{P}$ is in $\mathcal{MR}$.

From Proposition 3.26 it follows that the set $MR(\mathcal{P})$ can be computed by first transforming $\mathcal{P}$ into the program $\mathcal{P}^u \cup D^u$ and next by determining the arguments $p[i]$ of $\mathcal{P}^u$ such that $\{t_i \mid p(t_1, \ldots, t_n) \in \mathcal{MM}(\mathcal{P}^u \cup D^u)\}$ is finite. We call such arguments *limited in* $\mathcal{MM}(\mathcal{P}^u \cup D^u)$.

We will show that checking whether an argument of $\mathcal{P}^u$ is limited in $\mathcal{MM}(\mathcal{P}^u \cup D^u)$ is decidable. This comes from the fact that the transformed program $\mathcal{P}^u \cup D^u$ belongs to a class of logic programs called Datalog$_{nS}$. We will start by introducing the class of Datalog$_{nS}$ programs and its properties and then will exploit such properties to provide an algorithm able to compute the limited arguments in $\mathcal{MM}(\mathcal{P}^u \cup D^u)$.

Datalog [77] is the class of function-free logic programs, where predicates are partitioned into base and derived and the only terms are constants or variables, called *data terms*. Different extensions of Datalog have been studied in the literature, including programs with stratified and general negation, programs with disjunctive heads and programs with negation and disjunctive heads. It is well known that the complexity of computing the minimum model for Datalog programs is polynomial in the size of the input databases.

Datalog$_{nS}$ (*Datalog* with n successors), proposed in [29], is an extension of Datalog with a limited use of function symbols capable of representing infinite phenomena like flow of time, state transitions, construction of plans, etc. An example of a Datalog$_{nS}$ program is reported below.

Example 3.29. Consider the following program $\mathcal{P}_{3.29}$:

$$r: \ \mathtt{meets}(\mathtt{T}+\mathtt{1},\mathtt{Y}) \leftarrow \mathtt{follows}(\mathtt{X},\mathtt{Y}),\mathtt{meets}(\mathtt{T},\mathtt{X}).$$

where $T+1$ is a shorthand for $+1(T)$ and $+1$ is a function symbol. Rule r schedules the meetings of graduate students with their common advisor, where $\mathtt{meets}(\mathtt{t},\mathtt{x})$ *means that x meets her/his advisor in day t.* □

The problem of checking the finiteness of the minimum model of Datalog$_{nS}$ programs is decidable [29]. Predicates in Datalog$_{nS}$ can have an arbitrary number of unary function symbols and they can appear in one fixed argument.

This argument corresponds to a *state* (in Example 3.29 each *state* represents a particular moment of time), whereas function symbols map a state to another. Predicates containing a functional argument are called functional too. Functional arguments contain *functional terms*, which are built from a distinguished functional constant 0, a distinguished functional variable T and function symbols. For instance, in the program $\mathcal{P}_{3.29}$ of Example 3.29, terms 0, T and $T+1$ are functional terms.

Other syntactical restrictions of Datalog$_{nS}$ programs hold: i) rules are range restricted, ii) rule bodies are nonempty, iii) rules do not contain ground terms, and iv) functional terms in rules are of depth at most 1.

Datalog$_{1S}$ is a particular subclass of Datalog$_{nS}$ admitting exactly one unary function symbol (+1), so that functional ground terms can simply be seen as numbers representing time. For the sake of presentation, in the following we will briefly review the semantics of Datalog$_{1S}$ programs.

Example 3.30. Consider the program $\mathcal{P}_{3.30}$ obtained from $\mathcal{P}_{3.29}$ of Example 3.29 plus the rule:

$$\texttt{meets(T,Y)} \leftarrow \texttt{start(T,Y).}$$

and the following database $D_{3.30}$:

```
start(0, emma).
follows(emma, kathy).
follows(kathy, emma).
```

The minimal model $M_{3.30}$ of this program is composed by facts

```
follows(emma, kathy) follows(kathy, emma)
start(0, emma)       meets(0, emma)
```

and the following regularly repeating functional facts:

```
meets(1, kathy) meets(2, emma)
meets(3, kathy) meets(4, emma)
meets(5, kathy) meets(6, emma)
...             ...
```

where 1 is an abbreviation for 0+1, 2 is an abbreviation for (0+1)+1, and so on. □

Let $\mathcal{P} \cup D$ be a Datalog$_{1S}$ program, M be the model of $\mathcal{P} \cup D$ and t a ground functional term, the *state* $M[t]$ of M is $M[t] = \{p(\mathbf{a}) \mid p(t, \mathbf{a}) \in M\}$; the *snapshot* $M(t)$ of M is $M(t) = \{p(t, \mathbf{a}) \mid p(t, \mathbf{a}) \in M\}$; the *data part* M^d of M is the set of all the data facts in M. The *period* of M is a pair (t_1, t_2), where ground functional terms t_1 and t_2 are such that $t_1 < t_2$ and represent the smallest different times with the same state. It has been shown in [29] that $M[t_1 + k] = M[t_2 + k]$ for all $k \geq 0$.

Example 3.31. Consider the program $\mathcal{P}_{3.30}$ and the database $D_{3.30}$ from previous examples. Let $M_{3.30}$ be the minimal model of $\mathcal{P}_{3.30} \cup D_{3.30}$. Examples of state, snapshot and data part of $M_{3.30}$ are $M_{3.30}[0] = \{\texttt{start}(\texttt{emma}), \texttt{meets}(\texttt{emma})\}$, $M_{3.30}(0) = \{\texttt{start}(0, \texttt{emma}), \texttt{meets}(0, \texttt{emma})\}$ and $M^d = \{\texttt{follows}(\texttt{emma}, \texttt{kathy}), \texttt{follows}(\texttt{kathy}, \texttt{emma})\}$. Intuitively, $M_{3.30}$ repeats with period $(1, 3)$, i.e. $M_{3.30}[1 + k] = M_{3.30}[3 + k]$ for every $k \geq 0$. □

It has been shown in [29] that every Datalog$_{1S}$ program has a "periodic" minimal model and the finiteness of the model of a Datalog$_{1S}$ program $\mathcal{P} \cup D$ can be checked in polynomial space in the number of facts of D.

Observe that by construction, all predicates of $\mathcal{P}^u \cup D^u$ are unary and functional, the number of facts in D^u is equal to the number of arguments of $args(\bar{\mathcal{B}})$, and the number of function symbols in $\mathcal{P}$ and $\mathcal{P}^u \cup D^u$ coincide.

We consider two different cases on the base of whether the input program $\mathcal{P}$ contains only one or more than one function symbols, that is whether $\mathcal{P}^u \cup D^u$ is a Datalog$_{1S}$ or a Datalog$_{nS}$ program. Thus, in this section we present an algorithm computing the set of m-restricted arguments for a program $\mathcal{P}$ containing only one function symbol, i.e. $\mathcal{P}^u \cup D^u$ is a Datalog$_{1S}$ program.

We point out that, as the complexity of checking whether a Datalog$_{nS}$ program terminates may be higher than that of checking termination of a Datalog$_{1S}$ program, we could apply a less expensive (and less general) technique for checking program termintion, by considering a target program $\mathcal{P}^u \cup D^u$ where all function symbols are replaced by a single function symbol.

We start by introducing some definitions and results used hereafter to define the complexity of our algorithms.

The following lemma shows the relation between the syntactic size of a given program $\mathcal{P}$ and the syntactic size of the transformed program $\mathcal{P}^u \cup D^u$.

Lemma 3.32. *Given a program $\mathcal{P}$, $||\mathcal{P}^u \cup D^u|| = O(||\mathcal{P}||^2)$.*

Proof. By definition of $\mathcal{P}^u \cup D^u$, the number of facts in D^u is equal to the number of arguments of $args(\bar{\mathcal{B}})$ and the number of rules in $\mathcal{P}^u$ is at most $n \cdot a_p \cdot a_f$, where n is the number of rules, a_p is the maximum arity of predicates, and a_f is the maximum arity of function symbols in the program. Moreover, the maximum number of predicates in the body of rules in $\mathcal{P}^u$ is $p \cdot a_p$, where p is the maximum number of predicates in the body of rules. Finally, the maximum arity of predicates and function symbols of $\mathcal{P}^u$ is 1. Then, we have that $||\mathcal{P}^u|| = O((n \cdot a_p \cdot a_f) \cdot (p \cdot a_p)) = O(||\mathcal{P}||^2)$ and $||D^u|| = O(||\mathcal{P}||)$, consequently $||\mathcal{P}^u \cup D^u|| = O(||\mathcal{P}||^2)$. □

Programs with only one function symbol

The main function of the algorithm computing the set of m-restricted arguments for programs containing only one function symbol is *ComputeM-Restricted*. It takes as input a program $\mathcal{P}$ and returns as output the set of its m-restricted arguments.

Theorem 3.33. *For any program* $\mathcal{P}$, $MR(\mathcal{P}) = ComputeMRestricted(\mathcal{P})$. □

The function starts by computing the transformed program $\mathcal{P}^u \cup D^u$ (line 2). Next it computes the period (t_1, t_2) of the model of $\mathcal{P}^u \cup D^u$ (lines 3-15). In particular, since $\mathcal{P}^u \cup D^u$ is a unary Datalog$_{1S}$ program, the number of states of $\mathcal{P}^u \cup D^u$ is bounded by 2^{fsize}, where $fsize$ is the number of predicates in $\mathcal{P}^u \cup D^u$. Note that all arguments not limited in $M = \mathcal{MM}(\mathcal{P}^u \cup D^u)$ occur in predicates belonging to the states ranging from $M[t_1]$ to $M[t_2]$. Then, the function computes these states and deletes from the output set all the corresponding arguments (lines 16-21).

The computation of a state $M[t]$ of M is done by means of function *ComputeState*. It takes as input the transformation $\mathcal{P}^u \cup D^u$ of a program $\mathcal{P}$ and a ground term t and returns as output the state of the model of $\mathcal{P}^u \cup D^u$ evaluated in t. Computing a state $M[t]$ is performed by checking whether $\mathcal{P}^u \cup D^u \models p(t)$, for every predicate p occurring in $\mathcal{P}^u \cup D^u$. Function Models is in charge of checking whether $\mathcal{P}^u \cup D^u \models p(t)$ and it is a simplified version of the function proposed in [28], specific for unary programs with functional predicates only. This function is based on the following lemma: the notation $\mathcal{P}\{u\}$ denotes the program obtained by replacing every occurrence of the functional variable T in $\mathcal{P}$ with a ground functional term u.

Lemma 3.34. *[28] Let* $\mathcal{P} \cup D$ *be a Datalog*$_{1S}$ *program,* $Q(t, \boldsymbol{a})$ *a ground atomic query. Then,* M *is a model of* $\mathcal{P} \cup D \cup \neg Q(t, \boldsymbol{a})$ *iff the following conditions hold:*

```
input  : A positive normal program 𝒫.
output: The set MR(𝒫).
1:  MR(𝒫) := arg(𝒫);
    // Constructing 𝒫^u ∪ D^u with only one function symbol.
2:  𝒫^u ∪ D^u := Compute𝒫^u ∪ D^u(𝒫);
    // Computing the period.
3:  t_1 := 0;
4:  t_2 := 1;
5:  while true do
6:      M[t_2] := ComputeState(𝒫^u ∪ D^u, t_2);
7:      for t' := t_2-1 to 0 do
8:          M[t'] := ComputeState(𝒫^u ∪ D^u, t');
9:          if M[t'] = M[t_2] then
10:             t_1 = t';
11:             break while;
12:         end
13:     end
14:     t_2 := t_2 + 1;
15: end
    // Finding m-restricted arguments.
16: t* := t_1;
17: repeat
18:     M[t*] := ComputeState(𝒫^u ∪ D^u, t*);
19:     MR(𝒫) := MR(𝒫) − {p[i] | p_i() ∈ M[t*]};
20:     t* := t* + 1;
21: until t* = t_2;
22: return MR(𝒫);
```

Fig. 3.2: Function ComputeMRestricted

- $D \subseteq M$ and $Q(t, \mathbf{a}) \notin M(t)$;
- $M(u) \cup M(u+1) \cup M^d \models \mathcal{P}\{u\}$ for any ground functional term u. □

Let us start by presenting the complexity of function Models.

Proposition 3.35. *Let $\mathcal{P}^u \cup D^u$ be the transformation of a program $\mathcal{P}$ with one function symbol and $Q(t)$ be a ground atomic query, Function* Models *performs in polynomial space w.r.t. $||\mathcal{P}^u \cup D^u||$ and in polylogarithmic space w.r.t. $depth(t)$.*

Proof. The size of every state of the model M of $\mathcal{P}^u \cup D^u$ depends on the number of different ground atoms that can occur in one state; this number, denoted by $fsize$, is polynomial in $||\mathcal{P}^u \cup D^u||$. In a similar way, a snapshot

```
input  : The transformation P^u ∪ D^u of a program P.
         A ground atomic query Q(t).
output: Truth value of P^u ∪ D^u ⊨ Q(t).
 1: m := 2^{fsize};
 2: v := 0;
 3: CurSnap := Guess M(0);
 4: NextSnap := null;
 5: satisf := CurSnap ⊨ D^u ∪ ¬Q(t);
 6: while satisf and v < m do
 7:     NextSnap := Guess M(v + 1);
 8:     satisf := CurSnap ∪ NextSnap ⊨ P^u{v} ∪ ¬Q(t);
 9:     CurSnap := NextSnap;
10:     v := v + 1;
11: end
12: return not satisf;
```

Fig. 3.3: Function Models

$M(t)$ can be encoded as a pair $(t, M[t])$, requiring polynomial space w.r.t. $||\mathcal{P}^u \cup D^u||$ and logarithmic space w.r.t. $depth(t)$ (recall that t is a number that can be encoded in binary).

Function *Models* is a non deterministic algorithm which implements Lemma 3.34 with some simplifications due to the syntactical form of $\mathcal{P}^u \cup D^u$ (all predicates are unary and functional). The application of Lemma 3.34 consists in verifying whether $\mathcal{P}^u \cup D^u \cup \neg Q(t)$ admits a model, that is whether $\mathcal{P}^u \cup D^u \models Q(t)$. Moreover, it first guesses the initial snapshot of the minimal Herbrand model of $\mathcal{P}^u \cup D^u$. Guessing a snapshot is obviously space polynomial in $||\mathcal{P}^u \cup D^u||$ and logarithmic space in the depth of the given term. Verifying whether CurSnap $\models D^u \cup \neg Q(t)$ can be done in polynomial space w.r.t. $||\mathcal{P}^u \cup D^u||$ since it simply needs to check whether $D^u \subseteq$ CurSnap $\wedge\ Q(t) \notin$ CurSnap. The cycle in the algorithm performs at most m iterations, which is exponential in $||\mathcal{P}^u \cup D^u||$ but can be encoded in binary, requiring polynomial space. Moreover, the ground functional term v and the ground functional term t appearing in the query Q can be encoded in binary too, requiring a polynomial amount of memory for v in $||\mathcal{P}^u \cup D^u||$ (because $v < m$) and a logarithmic amount of space for t in $depth(t)$. Again, at each iteration, guessing the snapshot $M(v+1)$ is space polynomial in $||\mathcal{P}^u \cup D^u||$ and logarithmic space in $depth(v+1)$, but since $v < m$, the space for storing $v+1$ is at most polynomial in $||\mathcal{P}^u \cup D^u||$. Answering to CurSnap $\cup$ NextSnap $\models \mathcal{P}^u\{v\} \cup \neg Q(t)$ can be done in polynomial space in $||\mathcal{P}^u \cup D^u||$ and polylogarithmic space w.r.t.

```
    input  : The transformation P^u ∪ D^u of a program P.
             A ground functional term t.
    output: The state M[t].
 1: M[t] := ∅;
 2: foreach predicate p do
 3:     if Models (P^u ∪ D^u,p(t)) then
 4:         M[t] := M[t] ∪ {p()};
 5:     end
 6: end
 7: return M[t];
```

Fig. 3.4: Function ComputeState

$depth(t)$. Finally, by Savitch's theorem, every non deterministic space polynomial algorithm can be rewritten into a deterministic one which performs in quadratically more space. □

Since predicates in $\mathcal{P}^u \cup D^u$ are unary, the number of different atomic queries to be answered in Function *ComputeState* is polynomial in the size of the program. Then, computing a state has the same complexity of the Function *Models*.

Lemma 3.36. *Let $\mathcal{P}^u \cup D^u$ be the transformation of a program $\mathcal{P}$ with one function symbol. Computing the state of the model of $\mathcal{P}^u \cup D^u$ at the given time t requires polynomial space w.r.t. $||\mathcal{P}^u \cup D^u||$ and polylogarithmic space w.r.t. $depth(t)$.*

Proof. Straightforward from previous proposition and considerations. □

We can now present the main complexity result stating that *computing the set of m-restricted arguments of a program $\mathcal{P}$ with one function symbol is space polynomial w.r.t. $||\mathcal{P}||$.*

Theorem 3.37. *Given a program $\mathcal{P}$ containing only one function symbol, the complexity of computing $MR(\mathcal{P})$ is space polynomial w.r.t. $||\mathcal{P}||$.*

Proof. In Function *ComputeMRestricted*, `Compute`$\mathcal{P}^u \cup D^u(\mathcal{P})$ requires polynomial space in $||\mathcal{P}||$, from Lemma 3.32. The next phase of the algorithm computes the period of the model of $\mathcal{P}^u \cup D^u$ which is crucial for finding the m-restricted arguments of $\mathcal{P}$. The whole operation takes at most polynomial space w.r.t. $||\mathcal{P}^u \cup D^u||$ since by Lemma 3.36 `ComputeState` requires polynomial space in $||\mathcal{P}^u \cup D^u||$ and polylogarithmic space in $depth(t_2)$. Note that

$depth(t_2)$ is at most exponential in the maximum size of a state of $\mathcal{P}^u \cup D^u$ (i.e. $fsize$), then "polylogarithmic space in $depth(t_2)$" means polynomial space w.r.t. $||\mathcal{P}^u \cup D^u||$. Checking whether $M[t'] = M[t_2]$ requires obviously polynomial space in $||\mathcal{P}^u \cup D^u||$. Finally, storing variables t_1, t_2, t' requires polynomial space in $||\mathcal{P}^u \cup D^u||$. From Lemma 3.32, the whole phase requires polynomial space w.r.t. $||\mathcal{P}||$. The last phase computes the set $MR(\mathcal{P})$. From the previous considerations, the last phase requires polynomial space w.r.t. $||\mathcal{P}||$ too. □

Corollary 3.38. *Given a program $\mathcal{P}$, the complexity of checking whether $\mathcal{P} \in \mathcal{MR}$ is space polynomial w.r.t. $||\mathcal{P}||$ if $\mathcal{P}$ contains at most one function symbol.*

Proof. Straightforward from Theorem 3.37. □

Programs with more than one function symbol

So far we have considered programs with only one functions symbol. As said before, whenever programs contain more than one function symbol, we can perform a less accurate analysis, by replacing all function symbols with a unique symbol, even if they have different arities. The resulting unary program uses only one function symbol. This means that there could be mapping-restricted programs which are not recognized to be in $\mathcal{MR}$.

To enlarge the class of $\mathcal{MR}$ programs we can take into account the fact that programs may contain more than one function symbol rewriting them into a Datalog$_{nS}$ program. The counterpart of this growth of expressivity is obviously a greater computational complexity.

Indeed, as the complexity of checking whether the model of a Datalog$_{nS}$ program is finite is exponential, we would obtain that for any program $\mathcal{P}$ with more than one function symbol, the complexity of both computing $MR(\mathcal{P})$ and checking whether $\mathcal{P} \in \mathcal{MR}$ is at most time exponential w.r.t. $||\mathcal{P}||$.

3.5 Graph-based approaches

This section presents two approaches (Γ-acyclicity and Safe programs [17]) which rely on the definition of particular graphs constructed over a logic program. The idea behind these approaches is that if the graph constructed for a logic program $\mathcal{P}$ is acyclic, for some notion of acyclicity, then $\mathcal{P}$ is limited.

3.5.1 Γ-acyclicity

In this section we exploit the role of function symbols for checking limitedness of logic programs. In particular, the focus will be on the *termination* of normal positive programs $\mathcal{P}$ (thus, w.r.t. $pred(\mathcal{P})$). Recall that restricting the analysis to normal positive programs is w.l.o.g., as shown in Section 3.1.

We also assume that if the same variable X appears in two terms occurring in the head and body of a rule respectively, then at most one of the two terms is a complex term and that the nesting level of complex terms is at most one. There is no real restriction in such an assumption as every program could be rewritten into an equivalent program satisfying such a condition.

The following example shows a program admitting a finite minimum model, but, for example, the argument-restricted criterion is not able to detect it. Intuitively, the definition of argument restricted programs does not take into account the possible presence of different function symbols in the program that may prohibit the propagation of values in some rules and, consequently, guarantee the termination of the computation.

Example 3.39. Consider the following program $P_{3.39}$:

$$
\begin{aligned}
r_0 &: \mathtt{s(X)} \leftarrow \mathtt{b(X)}.\\
r_1 &: \mathtt{r(f(X))} \leftarrow \mathtt{s(X)}.\\
r_2 &: \mathtt{q(f(X))} \leftarrow \mathtt{r(X)}.\\
r_3 &: \mathtt{s(X)} \leftarrow \mathtt{q(g(X))}.
\end{aligned}
$$

The program is not argument-restricted since the argument ranking function ϕ_{min} cannot assign any value to r[1], q[1], and s[1]. However the computation always terminates, independently from the given set of facts. □

In order to represent the propagation of values among arguments, we introduce the concept of *labeled argument graphs*. Intuitively, it is an extension of the argument graph where each edge has a label describing how the term propagated from one argument to another changes. Arguments that are not dependent on a cycle can propagate a finite number of values and, therefore, are limited.

Thus, we can delete edges, from the labeled argument graph, ending in the nodes corresponding to limited arguments. Then, the resulting graph, called *propagation graph*, is deeply analyzed to identify further limited arguments.

Definition 3.40 (Labeled argument graph). *Let $\mathcal{P}$ be a program. The* labeled argument graph *$\mathcal{G}_L(\mathcal{P})$ is a labeled directed graph $(args(\mathcal{P}), E)$ where*

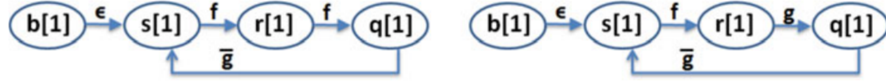


Fig. 3.5: Labeled argument graphs of programs $P_{3.39}$ (left) and $P_{3.41}$ (right)

E is a set of labeled edges defined as follows. For each pair of nodes $p[i], q[j] \in args(\mathcal{P})$ such that there is a rule r with $head(r) = p(v_1, ..., v_n)$, $q(u_1, ..., u_m) \in body(r)$, and terms u_j and v_i have a common variable X, there is an edge $(q[j], p[i], \alpha) \in E$ such that

- $\alpha = \epsilon$ *if* $u_j = v_i = X$,
- $\alpha = f$ *if* $u_j = X$ *and* $v_i = f(..., X, ...)$,
- $\alpha = \overline{f}$ *if* $u_j = f(..., X, ...)$ *and* $v_i = X$. □

In the definition above, the symbol ϵ denotes the empty label which concatenated to a string does not modify the string itself, that is, for any string s, $s\epsilon = \epsilon s = s$.

The labeled argument graph of program $P_{3.39}$ is shown in Figure 3.5 (left). The edges of this graph represent how the propagation of values occurs. For instance, edge $(\texttt{b}[1], \texttt{s}[1], \epsilon)$ states that a term `t` is propagated without changes from `b`[1] to `s`[1] if rule r_0 is applied; analogously, edge $(\texttt{s}[1], \texttt{r}[1], \texttt{f})$ states that starting from a term `t` in `s`[1] we obtain `f(t)` in `r`[1] if rule r_1 is applied, whereas edge $(\texttt{q}[1], \texttt{s}[1], \overline{\texttt{g}})$ states that starting from a term `g(t)` in `q`[1] we obtain `t` in `s`[1] if rule r_3 is applied.

Given a path π in $\mathcal{G}_L(\mathcal{P})$ of the form $(a_1, b_1, \alpha_1), ..., (a_m, b_m, \alpha_m)$, we denote with $\lambda(\pi)$ the string $\alpha_1 ... \alpha_m$. We say that π *spells a string* w if $\lambda(\pi) = w$. Intuitively, the string $\lambda(\pi)$ describes a sequence of function symbols used to compose and decompose complex terms during the propagation of values among the arguments in π.

Example 3.41. Consider program $P_{3.41}$ derived from program $P_{3.39}$ of Example 3.39 by replacing rule r_2 with the rule $\texttt{q(g(X))} \leftarrow \texttt{r(X)}$. The labeled argument graph $\mathcal{G}_L(P_{3.41})$ is reported in Figure 3.5 (right). Considering the cyclic path $\pi = (\texttt{s}[1], \texttt{r}[1], \texttt{f})$, $(\texttt{r}[1], \texttt{q}[1], \texttt{g})$, $(\texttt{q}[1], \texttt{s}[1], \overline{\texttt{g}})$, $\lambda(\pi) = \texttt{fg}\overline{\texttt{g}}$ represents the fact that starting from a term `t` in `s`[1] we may obtain the term `f(t)` in `r`[1], then we may obtain term `g(f(t))` in `q`[1], and term `f(t)` in `s`[1], and so on. Since we may obtain a larger term in `s`[1], the arguments depending on this cyclic path may not be limited.

Consider now program $P_{3.39}$, whose labeled argument graph is shown in Figure 3.5 (left), and the cyclic path $\pi' = (\mathtt{s}[1], \mathtt{r}[1], \mathtt{f})$, $(\mathtt{r}[1], \mathtt{q}[1], \mathtt{f})$, $(\mathtt{q}[1], \mathtt{s}[1], \overline{\mathtt{g}})$. Observe that starting from a term $\mathtt{t}$ in $\mathtt{s}[1]$ we may obtain term $\mathtt{f(t)}$ in $\mathtt{r}[1]$ (rule r_1), then we may obtain term $\mathtt{f(f(t))}$ in $\mathtt{q}[1]$ (rule r_2). At this point the propagation in this cyclic path terminates since the head atom of rule r_2 containing term $\mathtt{f(X)}$ cannot match with the body atom of rule r_3 containing term $\mathtt{g(X)}$. The string $\lambda(\pi') = \mathtt{ff}\overline{\mathtt{g}}$ represents the propagation described above. Observe that for this program all arguments are limited. □

Let π be a path from $p[i]$ to $q[j]$ in the labeled argument graph. Let $\hat{\lambda}(\pi)$ be the string obtained from $\lambda(\pi)$ by iteratively eliminating pairs of the form $\alpha\overline{\alpha}$ until the resulting string cannot be further reduced. If $\hat{\lambda}(\pi) = \epsilon$, then starting from a term t in $p[i]$ we obtain the same term t in $q[j]$. Consequently, if $\hat{\lambda}(\pi)$ is a non-empty sequence of function symbols $f_{i_1}, f_{i_2} ..., f_{i_k}$, then starting from a term t in $p[i]$ we may obtain a larger term in $q[j]$. For instance, if $k = 2$ and f_{i_1} and f_{i_2} are of arity one, we may obtain $f_{i_2}(f_{i_1}(t))$ in $q[j]$. Based on this intuition we introduce now a grammar $\Gamma_{\mathcal{P}}$ in order to distinguish the sequences of function symbols used to compose and decompose complex terms in a program $\mathcal{P}$, such that starting from a given term we obtain a larger term.

Recall that $F_{\mathcal{P}}$ denotes the set of all function symbols occurring in a program $\mathcal{P}$. We denote by $\overline{F}_{\mathcal{P}} = \{\overline{f} \mid f \in F_{\mathcal{P}}\}$ and $T_{\mathcal{P}} = F_{\mathcal{P}} \cup \overline{F}_{\mathcal{P}}$.

Definition 3.42. *Let $\mathcal{P}$ be a program, the* grammar $\Gamma_{\mathcal{P}}$ *is a 4-tuple $(N, T_{\mathcal{P}}, R, S)$, where $N = \{S, S_1, S_2\}$ is the set of nonterminal symbols, S is the start symbol, and R is the set of production rules defined below:*

1. $S \rightarrow S_1\, f_i\, S_2$, $\quad \forall f_i \in F_{\mathcal{P}}$;
2. $S_1 \rightarrow f_i\, S_1\, \overline{f}_i\, S_1 \mid \epsilon$, $\quad \forall f_i \in F_{\mathcal{P}}$;
3. $S_2 \rightarrow S_1\, S_2 \mid f_i\, S_2 \mid \epsilon$, $\quad \forall f_i \in F_{\mathcal{P}}$. □

The language $\mathcal{L}(\Gamma_{\mathcal{P}})$ is the set of strings generated by $\Gamma_{\mathcal{P}}$.

Example 3.43. Let $F_{\mathcal{P}} = \{\mathtt{f}, \mathtt{g}, \mathtt{h}\}$ be the set of function symbols occurring in a program $\mathcal{P}$. Then strings $\mathtt{f}$, $\mathtt{fg}\overline{\mathtt{g}}$, $\mathtt{g}\overline{\mathtt{g}}\mathtt{f}$, $\mathtt{fg}\overline{\mathtt{g}}\mathtt{h}\overline{\mathtt{h}}$, $\mathtt{fhg}\overline{\mathtt{g}}\overline{\mathtt{h}}$ belong to $\mathcal{L}(\Gamma_{\mathcal{P}})$ and represent, assuming that $\mathtt{f}$ is a unary function symbol, different ways to obtain term $\mathtt{f(t)}$ starting from term $\mathtt{t}$. □

Note that only if a path π spells a string $w \in \mathcal{L}(\Gamma_{\mathcal{P}})$, then starting from a given term in the first node of π we may obtain a larger term in the last node of π. Moreover, if this path is cyclic, then the arguments depending on

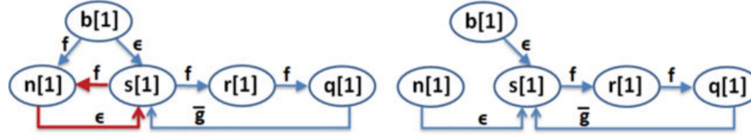


Fig. 3.6: Labeled argument graph (left) and propagation graph (right) of program $P_{3.45}$

it may not be limited. On the other hand, all arguments not depending on a cyclic path π spelling a string $w \in \mathcal{L}(\Gamma_{\mathcal{P}})$ are limited.

Given a program $\mathcal{P}$ and a set of arguments $\mathcal{A}$ recognized as limited by a specific criterion (eg. argument-restricted, mapping-restricted), the *propagation graph* of $\mathcal{P}$ with respect to $\mathcal{A}$, denoted by $\Delta(\mathcal{P}, \mathcal{A})$, consists of the subgraph derived from $\mathcal{G}_L(\mathcal{P})$ by deleting edges ending in a node of $\mathcal{A}$. Although we can consider any set $\mathcal{A}$ of limited arguments, in the following we assume $\mathcal{A} = AR(\mathcal{P})$ and, for the simplicity of notation, we denote $\Delta(\mathcal{P}, AR(\mathcal{P}))$ as $\Delta(\mathcal{P})$. Even if more general termination criteria have been defined in the literature, here we consider the AR criterion since it is the most general among those so far proposed having polynomial time complexity.

Definition 3.44 (Γ-acyclic Arguments and Γ-acyclic Programs). *Given a program $\mathcal{P}$, the set of its* Γ-acyclic arguments, *denoted by $\Gamma A(\mathcal{P})$, consists of all arguments of $\mathcal{P}$ not depending on a cyclic path in $\Delta(\mathcal{P})$ spelling a string of $\mathcal{L}(\Gamma_{\mathcal{P}})$. A program $\mathcal{P}$ is called* Γ-acyclic *if $\Gamma A(\mathcal{P}) = args(\mathcal{P})$, i.e. if there is no cyclic path in $\Delta(\mathcal{P})$ spelling a string of $\mathcal{L}(\Gamma_{\mathcal{P}})$. We denote the class of Γ-acyclic programs $\Gamma\mathcal{A}$.* □

Clearly, $AR(\mathcal{P}) \subseteq \Gamma A(\mathcal{P})$, i.e. the set of restricted arguments is contained in the set of Γ-acyclic arguments. As a consequence, the set of argument-restricted programs is a subset of the set of Γ-acyclic programs. Moreover, the containment is strict, as there exist programs that are Γ-acyclic, but not argument-restricted. For instance, program $P_{3.39}$ from Example 3.39 is Γ-acyclic, but not argument-restricted. Indeed, all cyclic paths in $\Delta(P_{3.39})$ do not spell strings belonging to the language $\mathcal{L}(\Gamma_{P_{3.39}})$.

The importance of considering the propagation graph instead of the labeled argument graph in Definition 3.44 is shown in the following example.

Example 3.45. Consider program $P_{3.45}$ below obtained from $P_{3.39}$ by adding rules r_4 and r_5.

$$r_0 : \mathtt{s(X)} \leftarrow \mathtt{b(X)}.$$
$$r_1 : \mathtt{r(f(X))} \leftarrow \mathtt{s(X)}.$$
$$r_2 : \mathtt{q(f(X))} \leftarrow \mathtt{r(X)}.$$
$$r_3 : \mathtt{s(X)} \leftarrow \mathtt{q(g(X))}.$$
$$r_4 : \mathtt{n(f(X))} \leftarrow \mathtt{s(X)}, \mathtt{b(X)}.$$
$$r_5 : \mathtt{s(X)} \leftarrow \mathtt{n(X)}.$$

The corresponding labeled argument graph $\mathcal{G}_L(P_{3.45})$ and propagation graph $\Delta(P_{3.45})$ are reported in Figure 3.6. Observe that arguments $\mathtt{n}[1]$ and $\mathtt{s}[1]$ are involved in the red cycle in the labeled argument graph $\mathcal{G}_L(P_{3.45})$ spelling a string of $\mathcal{L}(\Gamma_{P_{3.45}})$. At the same time this cycle is not present in the propagation graph $\Delta(P_{3.45})$ since $AR(P_{3.45}) = \{\mathtt{b}[1], \mathtt{n}[1]\}$ and the program is Γ-acyclic. □

Theorem 3.46. *Given a program $\mathcal{P}$,*

1. *all arguments in $\Gamma A(\mathcal{P})$ are limited;*
2. *if $\mathcal{P}$ is Γ-acyclic, then $\mathcal{P}$ is terminating.*

Proof. 1) As previously recalled, arguments in $AR(\mathcal{P})$ are limited. Let us now show that all arguments in $\Gamma A(\mathcal{P}) \setminus AR(\mathcal{P})$ are limited too. Suppose by contradiction that $q[k] \in \Gamma A(\mathcal{P}) \setminus AR(\mathcal{P})$ is not limited. Observe that depth of terms that may occur in $q[k]$ depends on the paths in the propagation graph $\Delta(\mathcal{P})$ that ends in $q[k]$. In particular, this depth may be infinite only if there is a path π from an argument $p[i]$ to $q[k]$ (not necessarily distinct from $p[i]$), such that $\hat{\lambda}(\pi)$ is a string of an infinite length composed by symbols in F_P. But this is possible only if this path contains a cycle spelling a string in $\mathcal{L}(\Gamma_{\mathcal{P}})$. Thus we obtain contradiction with Definition 3.44.
2) From the previous proof, it follows that every argument in the Γ-acyclic program can take values only from a finite domain. Consequently, the set of all possible derived ground terms is finite and every Γ-acyclic program is terminating. □

From the previous theorem we can also conclude that all Γ-acyclic programs admit a finite minimum model.

We conclude by observing that since the language $\mathcal{L}(\Gamma_{\mathcal{P}})$ is context-free, the analysis of paths spelling strings in $\mathcal{L}(\Gamma_{\mathcal{P}})$ can be carried out using pushdown automata.

As $\Gamma_{\mathcal{P}}$ is context free, the language $\mathcal{L}(\Gamma_{\mathcal{P}})$ can be recognized by means of a pushdown automaton $M = (\{q_0, q_F\}, T_{\mathcal{P}}, \Lambda, \delta, q_0, Z_0, \{q_F\})$, where q_0 is the initial state, q_F is the final state, $\Lambda = \{Z_0\} \cup \{F_i | f_i \in F_{\mathcal{P}}\}$ is the stack alphabet, Z_0 is the initial stack symbol, and δ is the transition function defined as follows:

1. $\delta(q_0, f_i, Z_0) = (q_F, F_i Z_0), \quad \forall f_i \in F_{\mathcal{P}}$,
2. $\delta(q_F, f_i, F_j) = (q_F, F_i F_j), \quad \forall f_i \in F_{\mathcal{P}}$,
3. $\delta(q_F, \overline{f}_j, F_j) = (q_F, \epsilon), \quad \forall f_i \in F_{\mathcal{P}}$.

The input string is recognized if after having scanned the entire string the automaton is in state q_F and the stack contains at least one symbol F_i.

A path π is called:

- *increasing*, if $\hat{\lambda}(\pi) \in \mathcal{L}(\Gamma_{\mathcal{P}})$,
- *flat*, if $\hat{\lambda}(\pi) = \epsilon$,
- *failing*, otherwise.

It is worth noting that $\lambda(\pi) \in \mathcal{L}(\Gamma_{\mathcal{P}})$ iff $\hat{\lambda}(\pi) \in \mathcal{L}(\Gamma_{\mathcal{P}})$ as function $\hat{\lambda}$ emulates the pushdown automaton used to recognize $\mathcal{L}(\Gamma_{\mathcal{P}})$. More specifically, for any path π and relative string $\lambda(\pi)$ we have that:

- if π is increasing, then the pushdown automaton recognizes the string $\lambda(\pi)$ in state q_F and the stack contains a sequence of symbols corresponding to the symbols in $\hat{\lambda}(\pi)$ plus the initial stack symbol Z_0;
- if π is flat, then the pushdown automaton does not recognize the string $\lambda(\pi)$; moreover, the entire input string is scanned, but the stack contains only the symbol Z_0;
- if $\hat{\lambda}(\pi)$ is failing, then the pushdown automaton does not recognize the string $\lambda(\pi)$ as it goes in an error state.

Complexity

Concerning the complexity of checking whether a program is Γ-acyclic, we first introduce definitions and results that will be used hereafter. We first introduce a tighter bound for the complexity of computing $AR(\mathcal{P})$.

Proposition 3.47. *For any program $\mathcal{P}$, the time complexity of computing $AR(\mathcal{P})$ is bounded by $O(|args(\mathcal{P})|^3)$.*

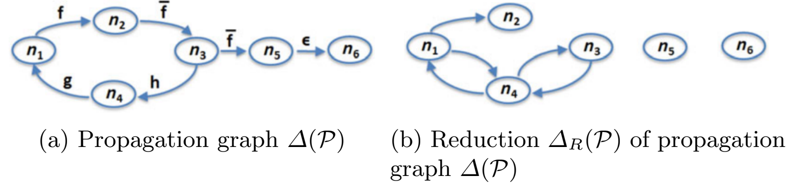
Proof. Assume that $n = |args(\mathcal{P})|$ is the total number of arguments of $\mathcal{P}$. First, it is important to observe the connection between the behavior of operator Ω and the structure of the labeled argument graph $\mathcal{G}_L(\mathcal{P})$. In particular, if the applications of the operator Ω change the rank of an argument $q[i]$ from 0 to k, then there is a path from an argument to $q[i]$ in $\mathcal{G}_L(\mathcal{P})$, where the number of edges labeled with some positive function symbol minus the number of edges labeled with some negative function symbol is at least k. Given a cycle in a labeled argument graph, let us call it *affected* if the number of edges labeled with some positive function symbol is greater than the number of edges labeled with some negative function symbol.

If an argument is not restricted, it is involved in or depends on an affected cycle. On the other hand, if after an application of Ω the rank assigned to an argument exceeds n, this argument is not restricted [60]. Recall that we are assuming that $d_{max} = 1$, where d_{max} is the largest depth of terms occurring in the heads of rules of $\mathcal{P}$ and, therefore, $M = n \times d_{max} = n$.

Now let us show that after $2n^2 + n$ iterations of Ω all not restricted arguments exceed rank n. Consider an affected cycle and suppose that it contains k arguments, whereas the number of arguments depending on this cycle, but not belonging to it is m. Obviously, $k + m \leq n$. All arguments involved in this cycle change their rank by at least one after k iterations of Ω. Thus their ranks will be greater than $n + m$ after $(n + m + 1) * k$ iterations. The arguments depending on this cycle, but not belonging to it, need at most another m iterations to reach the rank greater than n. Thus all unrestricted arguments exceed the rank n in $(n + m + 1) * k + m$ iterations of Ω. Since $(n + m + 1) * k + m = nk + mk + (k + m) \leq 2n^2 + n$, the restricted arguments are those that at step $2n^2 + n$ do not exceed rank n. It follows that the complexity of computing $AR(\mathcal{P})$ is bounded by $O(n^3)$ because we have to do $O(n^2)$ iterations and, for each iteration we have to check the rank of n arguments. □

In order to study the complexity of computing Γ-acyclic arguments of a program we introduce a directed (not labeled) graph obtained from the propagation graph.

Definition 3.48 (Reduction of $\Delta(\mathcal{P})$). *Given a program $\mathcal{P}$, the* reduction *of $\Delta(\mathcal{P})$ is a directed graph $\Delta_R(\mathcal{P})$ whose nodes are the arguments of $\mathcal{P}$ and there is an edge $(p[i], q[j])$ in $\Delta_R(\mathcal{P})$ iff there is a path π from $p[i]$ to $q[j]$ in $\Delta(\mathcal{P})$ such that $\hat{\lambda}(\pi) \in F_{\mathcal{P}}$.* □



(a) Propagation graph $\Delta(\mathcal{P})$ (b) Reduction $\Delta_R(\mathcal{P})$ of propagation graph $\Delta(\mathcal{P})$

Fig. 3.7: Propagation and reduction graphs

The reduction $\Delta_R(\mathcal{P})$ of the propagation graph $\Delta(\mathcal{P})$ from Figure 3.7a is shown in Figure 3.7b. It is simple to note that for each path in $\Delta(\mathcal{P})$ from node $p[i]$ to node $q[j]$ spelling a string of $\mathcal{L}(\Gamma_{\mathcal{P}})$ there exists a path from $p[i]$ to $q[j]$ in $\Delta_R(\mathcal{P})$ and vice versa. As shown in the lemma below, this property always holds.

Lemma 3.49. *Given a program $\mathcal{P}$ and arguments $p[i], q[j] \in args(\mathcal{P})$, there exists a path in $\Delta(\mathcal{P})$ from $p[i]$ to $q[j]$ spelling a string of $\mathcal{L}(\Gamma_{\mathcal{P}})$ iff there is a path from $p[i]$ to $q[j]$ in $\Delta_R(\mathcal{P})$.*

Proof. ($\Rightarrow$) Suppose there is a path π from $p[i]$ to $q[j]$ in $\Delta(\mathcal{P})$ such that $\lambda(\pi) \in \mathcal{L}(\Gamma_{\mathcal{P}})$. Then $\hat{\lambda}(\pi)$ is a non-empty string, say $f_1...f_k$, where $f_i \in F_{\mathcal{P}}$ for $i \in [1..k]$. Consequently, π can be seen as a sequence of subpaths $\pi_1, ..., \pi_k$, such that $\hat{\lambda}(\pi_i) = f_i$ for $i \in [1..k]$. Thus, from the definition of the reduction of $\Delta(\mathcal{P})$, there is a path from $p[i]$ to $q[j]$ in $\Delta_R(\mathcal{P})$ whose length is equal to $|\hat{\lambda}(\pi)|$.
($\Leftarrow$) Suppose there is a path $(n_1, n_2)...(n_k, n_{k+1})$ from n_1 to n_{k+1} in $\Delta_R(\mathcal{P})$. From the definition of the reduction of $\Delta(\mathcal{P})$, for each edge (n_i, n_{i+1}) there is a path, say π_i, from n_i to n_{i+1} in $\Delta(\mathcal{P})$ such that $\hat{\lambda}(\pi_i) \in F_{\mathcal{P}}$. Consequently, there is a path from n_1 to n_{k+1} in $\Delta(\mathcal{P})$, obtained as a sequence of paths $\pi_1, ..., \pi_k$ whose string is simply $\lambda(\pi_1)...\lambda(\pi_k)$. Since $\hat{\lambda}(\pi_i) \in F_{\mathcal{P}}$ implies that $\lambda(\pi_i) \in \mathcal{L}(\Gamma_{\mathcal{P}})$, for every $1 \leq i \leq k$, we have that $\lambda(\pi_1)...\lambda(\pi_k)$ belongs also to $\mathcal{L}(\Gamma_{\mathcal{P}})$. □

Proposition 3.50. *Given a program $\mathcal{P}$, the time complexity of computing the reduction $\Delta_R(\mathcal{P})$ is bounded by $O(|args(\mathcal{P})|^3 \times |F_{\mathcal{P}}|)$.*

Proof. The construction of $\Delta_R(\mathcal{P})$ can be performed as follows. First, we compute all the paths π in $\Delta(\mathcal{P})$ such that $|\hat{\lambda}(\pi)| \leq 1$. To do so, we use a slight variation of the Floyd-Warshall's transitive closure of $\Delta(\mathcal{P})$ which is defined by the following recursive formula. Assume that each node of $\Delta(\mathcal{P})$

is numbered from 1 to $n = |args(\mathcal{P})|$, then we denote with $path(i, j, \alpha, k)$ the existence of a path π from node i to node j in $\Delta(\mathcal{P})$ such that $\hat{\lambda}(\pi) = \alpha$, $|\alpha| \leq 1$ and π may go only through nodes in $\{1, ..., k\}$ (except for i and j).

The set of atoms $path(i, j, \alpha, k)$, for all values $1 \leq i, j \leq n$, can be derived iteratively as follows:

- (base case: $k = 0$) $path(i, j, \alpha, 0)$ holds if there is an edge (i, j, α) in $\Delta(\mathcal{P})$,
- (inductive case: $0 < k \leq n$) $path(i, j, \alpha, k)$ holds if
  - $path(i, j, \alpha, k-1)$ holds, or
  - $path(i, k, \alpha_1, k-1)$ and $path(k, j, \alpha_2, k-1)$ hold, $\alpha = \alpha_1\alpha_2$ and $|\alpha| \leq 1$.

Note that in order to compute all the possible atoms $path(i, j, \alpha, k)$, we need to first initialize every base atom $path(i, j, \alpha, 0)$ with cost bounded by $O(n^2 \times |F_{\mathcal{P}}|)$, as this is the upper bound for the number of edges in $\Delta(\mathcal{P})$. Then, for every $1 \leq k \leq n$, we need to compute all paths $path(i, j, \alpha, k)$, thus requiring a cost bounded by $O(n^3 \times |F_{\mathcal{P}}|)$ operations. The whole procedure will require $O(n^3 \times |F_{\mathcal{P}}|)$ operations. Since we have computed all possible paths π in $\Delta(\mathcal{P})$ such that $|\hat{\lambda}(\pi)| \leq 1$, we can obtain all the edges (i, j) of $\Delta_R(\mathcal{P})$ (according to Definition 3.48) by simply selecting the atoms $path(i, j, \alpha, k)$ with $\alpha \in F_{\mathcal{P}}$, whose cost is bounded by $O(n^2 \times |F_{\mathcal{P}}|)$. Then, the time complexity of constructing $\Delta_R(\mathcal{P})$ is $O(n^3 \times |F_{\mathcal{P}}|)$. □

Theorem 3.51. *The complexity of deciding whether a program $\mathcal{P}$ is Γ-acyclic is bounded by $O(|args(\mathcal{P})|^3 \times |F_{\mathcal{P}}|)$.*

Proof. Assume that $n = |args(\mathcal{P})|$ is the total number of arguments of $\mathcal{P}$. To check whether $\mathcal{P}$ is Γ-acyclic it is sufficient to first compute the set of restricted arguments $AR(\mathcal{P})$ which requires time $O(n^3)$ from Proposition 3.47. Then, we need to construct the propagation graph $\Delta(\mathcal{P})$, for which the maximum number of edges is $n^2 \times (|F_{\mathcal{P}}| + |\overline{F}_{\mathcal{P}}| + 1)$, then it can be constructed in time $O(n^2 \times |F_{\mathcal{P}}|)$ (recall that we are not taking into account the cost of scanning and storing the program). Moreover, starting from $\Delta(\mathcal{P})$, we need to construct $\Delta_R(\mathcal{P})$, which requires time $O(n^3 \times |F_{\mathcal{P}}|)$ (cf. Proposition 3.50) and then, following Lemma 3.49, we need to check whether $\Delta_R(\mathcal{P})$ is acylic. Verifying whether $\Delta_R(\mathcal{P})$ is acyclic can be done by means of a simple traversal of $\Delta_R(\mathcal{P})$ and checking if a node is visited more than once. The complexity of a depth-first traversal of a graph is well-known to be $O(|E|)$ where E is the set of edges of the graph. Since the maximum number of edges of $\Delta_R(\mathcal{P})$ is by

definition $n^2 \times |F_{\mathcal{P}}|$, the traversal of $\Delta_R(\mathcal{P})$ can be done in time $O(n^2 \times |F_{\mathcal{P}}|)$. Thus, the whole time complexity is still bounded by $O(n^3 \times |F_{\mathcal{P}}|)$. $\square$

Corollary 3.52. *For any program $\mathcal{P}$, the time complexity of computing $\Gamma A(\mathcal{P})$ is bounded by $O(|args(\mathcal{P})|^3 \times |F_{\mathcal{P}}|)$.*

Proof. Straightforward from the proof of Theorem 3.51. $\square$

As shown in the previous theorem, the time complexity of checking whether a program $\mathcal{P}$ is Γ-acyclic is bounded by $O(|args(\mathcal{P})|^3 \times |F_{\mathcal{P}}|)$, which is strictly related to the complexity of checking whether a program is argument-restricted, which is $O(|args(\mathcal{P})|^3)$. In fact, the new proposed criterion performs a more accurate analysis on how terms are propagated from the body to the head of rules by taking into account the function symbols occurring in such terms. Moreover, if a logic program $\mathcal{P}$ has only one function symbol, the time complexity of checking whether $\mathcal{P}$ is Γ-acyclic is the same as the one required to check if it is argument-restricted.

3.5.2 Safe programs

The Γ-acyclicity termination criterion presents some limitations, since it is not able to detect when a rule can be fired only a finite number of times during the evaluation of the program. The next example shows a simple terminating program which is not recognized by the Γ-acyclicity termination criterion.

Example 3.53. Consider the following logic program $P_{3.53}$:

$$
\begin{aligned}
r_1 &: \mathtt{p(X,X)} \leftarrow \mathtt{b(X)}.\\
r_2 &: \mathtt{p(f(X),g(X))} \leftarrow \mathtt{p(X,X)}.
\end{aligned}
$$

As the program is standard, it has a (finite) unique minimal model, which can can be derived using the immediate consequence operator. Moreover, independently from the set of facts D, the minimum model of $P_{3.53}$ is finite and its computation terminates. $\square$

Observe that the rules of program $P_{3.53}$ can be fired at most n times, where n is the number of ground b-atoms in any given set of facts D. Indeed, the recursive rule r_2 cannot fire itself since the newly generated atom is of the form $\mathtt{p(f(\cdot),g(\cdot))}$ and does not unify with its body.
As another example consider the recursive rule $\mathtt{q(f(X))} \leftarrow \mathtt{q(X)}, \mathtt{t(X)}$ and the

rule $\texttt{p(f(X),g(Y))} \leftarrow \texttt{p(X,Y),t(X)}$ where $\texttt{t[1]}$ is a limited argument. The firing of these rules is limited by the cardinality of the terms occurring in $\texttt{t[1]}$.

Thus, in this section, in order to define a more general termination criterion we introduce the *safety* function which, by detecting rules that can be executed only a finite number of times, derives a larger set of limited arguments of the program. We start by analyzing how rules may fire each other.

Definition 3.54 (Firing Graph). *Let $\mathcal{P}$ be a program and let r_1 and r_2 be (not necessarily distinct) rules of $\mathcal{P}$. We say that r_1* fires *r_2 iff head(r_1) and an atom in body(r_2) unify. The* firing graph *$\Sigma(\mathcal{P}) = (\mathcal{P}, E)$ consists of the set of nodes denoting the rules of $\mathcal{P}$ and the set of edges (r_i, r_j), with $r_i, r_j \in \mathcal{P}$, such that r_i fires r_j.* □

Example 3.55. Consider program $P_{3.53}$ of Example 3.53. The firing graph of this program contains two nodes r_1 and r_2 and an edge from r_1 to r_2. Rule r_1 fires rule r_2 as the head atom $\texttt{p(X,X)}$ of r_1 unifies with the body atom $\texttt{p(X,X)}$ of r_2. Intutively, this means that the execution of the first rule may cause the second rule to be fired. In fact, the execution of r_1 starting from the set of facts instance $D = \{\texttt{b(a)}\}$ produces the new atom $\texttt{p(a,a)}$. The presence of this atom allows the second rule to be fired, since the body of r_2 can be made true by means of the atom $\texttt{p(a,a)}$, producing the new atom $\texttt{p(f(a),g(a))}$. It is worth noting that the second rule cannot fire itself since *head*(r_2) does not unify with the atom $\texttt{p(X,X)}$ in *body*(r_2). □

The firing graph shows how rules may fire each other, and, consequently, the possibility to propagate values from one rule to another. Clearly, the number of terms occurring in an argument $p[i]$ can be infinite only if p is the head predicate of a rule that may be fired an infinite number of times. A rule may be fired an infinite number of times only if it depends on a cycle of the firing graph. Therefore, a rule not depending on a cycle can only propagate a finite number of values into its head arguments. Another important aspect is the structure of rules and the presence of limited arguments in their body and head atoms. As discussed at the beginning of this section, rules $\texttt{q(f(X))} \leftarrow \texttt{q(X),t(X)}$ and $\texttt{p(f(X),g(Y))} \leftarrow \texttt{p(X,Y),t(X)}$, where $\texttt{t[1]}$ is a limited argument, can be fired only a finite number of times. In fact, as variable $\texttt{X}$ in both rules can be substituted only by the terms occurring in $\texttt{t[1]}$, which are finite in number, the number of terms in $\texttt{q[1]}$ and $\texttt{p[1]}$ is finite as well, i.e. $\texttt{q[1]}$ and $\texttt{p[1]}$ are limited arguments. Since $\texttt{q[1]}$ is limited, the first rule can be applied only a finite number of times. In the second rule we have predicate $\texttt{p}$ of arity

two in the head, and we know that p[1] is a limited argument. Furthermore, note that in the second rule the domains of both head arguments p[1] and p[2] grow together each time this rule is applied. This behaviour will be soon formalized by the notion of *direct rule*. Consequently, the number of terms in p[2] must be finite as well as the number of terms in p[1] and this rule can be applied only a finite number of times.

We now introduce the notions of *direct rules* and *limited term*, that will be used to define a function, called *safety function*, that takes as input a set of limited arguments and derives a new set of limited arguments in $\mathcal{P}$.

Definition 3.56 (Direct rule). *Given a program $\mathcal{P}$ and a rule $r \in \mathcal{P}$, we say that r is* direct *if there exists an atom B in $body(r)$, denoted as $drb(r)$, such that:*

- *B is the only atom in $body(r)$ such that $pred(B)$ is mutually recursive with $pred(head(r))$;*
- *$pred(B) = pred(head(r))$;*
- *every other rule $r' \in \mathcal{P}$ with $pred(head(r')) = pred(head(r))$ has no body atom B' such that $pred(B')$ is mutually recursive with $pred(head(r'))$.* □

Intuitively, the domains of the head arguments of a direct rule change together each time this rule is fired.

Definition 3.57 (Limited terms). *Given a rule $r = q(t_1, ..., t_m) \leftarrow body(r) \in \mathcal{P}$ and a set $\mathcal{A}$ of limited arguments, we say that t_i is* limited *in r (or r* limits *t_i) w.r.t. $\mathcal{A}$ if one of the following conditions holds:*

1. *every variable X appearing in t_i also appears in an argument in $body(r)$ belonging to $\mathcal{A}$, or*
2. *r is a direct rule such that:*
   a) *for every atom $p(u_1, ..., u_n) \in head(r) \cup \{drb(r)\}$, all terms $u_1, ..., u_n$ are either simple or complex;*
   b) *the set of variables in $head(r)$ and $drb(r)$ coincide,*
   c) *there is an argument $q[j] \in A$.* □

Definition 3.58 (Safety Function). *For any program $\mathcal{P}$, let $\mathcal{A}$ be a set of limited arguments of $\mathcal{P}$ and let $\Sigma(\mathcal{P})$ be the firing graph of $\mathcal{P}$. The* safety function *$\Psi(\mathcal{A})$ denotes the set of arguments $q[i] \in args(\mathcal{P})$ such that for all rules $r = q(t_1, ..., t_m) \leftarrow body(r) \in \mathcal{P}$, either r does not depend on a cycle π of $\Sigma(\mathcal{P})$ or t_i is limited in r w.r.t. $\mathcal{A}$.* □

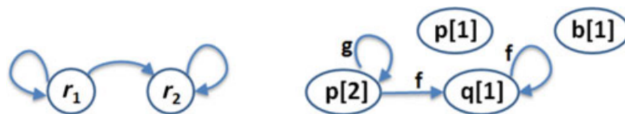


Fig. 3.8: Firing (left) and propagation (right) graphs of program $P_{3.59}$.

Example 3.59. Consider the following program $P_{3.59}$:

$$r_1: \ \mathtt{p(f(X), g(Y)) \leftarrow p(X, Y), b(X).}$$
$$r_2: \ \mathtt{q(f(Y)) \leftarrow p(X, Y), q(Y).}$$

Let $A = \Gamma A(\mathcal{P}) = \{\mathtt{b}[1], \mathtt{p}[1]\}$. The firing and the propagation graphs of this program are reported in Figure 3.8. The application of the safety function to the set of limited arguments $\mathcal{A}$ gives $\Psi(\mathcal{A}) = \{\mathtt{b}[1], \mathtt{p}[1], \mathtt{p}[2]\}$. Indeed:

- `b[1]` $\in \Psi(\mathcal{A})$ since `b` does not appear in the head of any rule; consequently all the rules with `b` in the head (i.e. the empty set) trivially satisfy the conditions of Definition 3.58.
- `p[1]` $\in \Psi(\mathcal{A})$ because the unique rule with `p` in the head (i.e. r_1) satisfies the first condition of Definition 3.57, that is, r_1 limits the term `f(X)` w.r.t. $\mathcal{A}$ in the head of rule r_1 corresponding to argument $\mathtt{p}[1]$.
- Since r_1 is direct and the second condition of Definition 3.57 is satisfied, `p[2]` $\in \Psi(\mathcal{A})$ as well. □

The following proposition shows that the safety function can be used to derive further limited arguments.

Proposition 3.60. *Let $\mathcal{P}$ be a program and let $\mathcal{A}$ be a set of limited arguments of $\mathcal{P}$. Then, all arguments in $\Psi(\mathcal{A})$ are also limited.*

Proof. Consider an argument $q[i] \in \Psi(\mathcal{A})$, then for every rule $r = q(t_1, ..., t_n) \leftarrow$ *body*(r) either r does not depend on a cycle of $\Sigma(\mathcal{P})$ or t_i is limited in r w.r.t. $\mathcal{A}$.

Clearly, if r does not depend on a cycle of $\Sigma(\mathcal{P})$, it can be fired a finite number of times as it is not 'effectively recursive' and does not depend on rules which are effectively recursive.

Moreover, if t_i is limited in r w.r.t. $\mathcal{A}$, we have that either:
1) The first condition of Definition 3.57 is satisfied (i.e. every variable X appearing in t_i also appears in an argument in *body*(r) belonging to $\mathcal{A}$). This

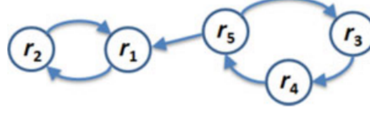


Fig. 3.9: Firing Graph of program $P_{3.61}$

means that variables in t_i can be replaced by a finite number of values.
2) The second condition of Definition 3.57 is satisfied. Let $p(t_1, ..., t_n) = head(r)$, the condition that all terms $t_1, ..., t_n$ must be simple or complex guarantees that, if terms in $head(r)$ grow, then they grow all together (conditions 2.a and 2.b). Moreover, if the growth of a term t_j is blocked (Condition 2.c), the growth of all terms (including t_i) is blocked too.
Therefore, if one of the two condition is satisfied for all rules defining q, the number of terms in $q[i]$ is finite. □

Unfortunately, as shown in the following example, the relationship $\mathcal{A} \subseteq \Psi(\mathcal{A})$ does not always hold for a generic set of arguments $\mathcal{A}$, even if the arguments in $\mathcal{A}$ are limited.

Example 3.61. Consider the following program $P_{3.61}$:

$$r_1 : \mathtt{p(f(X), Y)} \leftarrow \mathtt{q(X), r(Y)}.$$
$$r_2 : \mathtt{q(X)} \leftarrow \mathtt{p(X, Y)}.$$
$$r_3 : \mathtt{t(Y)} \leftarrow \mathtt{r(Y)}.$$
$$r_4 : \mathtt{s(Y)} \leftarrow \mathtt{t(Y)}.$$
$$r_5 : \mathtt{r(Y)} \leftarrow \mathtt{s(Y)}.$$

Its firing graph $\Sigma(P_{3.61})$ is shown in Figure 3.9, whereas the set of restricted arguments is $AR(P_{3.61}) = \Gamma A(P_{3.61}) = \{\mathtt{r}[1], \mathtt{t}[1], \mathtt{s}[1], \mathtt{p}[2]\}$. Considering the set $A = \{\mathtt{p}[2]\}$, we have that the safety function $\Psi(\{\mathtt{p}[2]\}) = \varnothing$. Therefore, the relation $\mathcal{A} \subseteq \Psi(\mathcal{A})$ does not hold for $\mathcal{A} = \{\mathtt{p}[2]\}$.

Moreover, regarding the set $\mathcal{A}' = \Gamma A(P_{3.61}) = \{\mathtt{r}[1], \mathtt{t}[1], \mathtt{s}[1], \mathtt{p}[2]\}$, we have $\Psi(\mathcal{A}') = \{\mathtt{r}[1], \mathtt{t}[1], \mathtt{s}[1], \mathtt{p}[2]\} = \mathcal{A}'$, i.e. the relation $\mathcal{A}' \subseteq \Psi(\mathcal{A}')$ holds. □

The following proposition states that if we consider the set $\mathcal{A}$ of Γ-acyclic arguments of a given program $\mathcal{P}$, the relation $\mathcal{A} \subseteq \Psi(\mathcal{A})$ holds.

Proposition 3.62. *For any logic program $\mathcal{P}$:*

1. $\Gamma A(\mathcal{P}) \subseteq \Psi(\Gamma A(\mathcal{P}))$;

2. $\Psi^i(\Gamma A(\mathcal{P})) \subseteq \Psi^{i+1}(\Gamma A(\mathcal{P}))$ *for* $i > 0$.

Proof. 1) Suppose that $q[k] \in \Gamma A(\mathcal{P})$. Then $q[k] \in AR(\mathcal{P})$ or $q[k]$ does not depend on a cycle in $\Delta(\mathcal{P})$ spelling a string of $\mathcal{L}(\Gamma_{\mathcal{P}})$. In both cases $q[k]$ can depend only on arguments in $\Gamma A(\mathcal{P})$. If $q[k]$ does not depend on any argument, then it does not appear in the head of any rule and, consequently, $q[k] \in \Psi(\Gamma A(\mathcal{P}))$. Otherwise, the first condition of Definition 3.57 is satisfied and $q[k] \in \Psi(\Gamma A(\mathcal{P}))$.
2) We prove that $\Psi^i(\Gamma A(\mathcal{P})) \subseteq \Psi^{i+1}(\Gamma A(\mathcal{P}))$ for $i > 0$ by induction. We start by showing that $\Psi^i(\Gamma A(\mathcal{P})) \subseteq \Psi^{i+1}(\Gamma A(\mathcal{P}))$ for $i = 1$, i.e. that the relation $\Psi(\Gamma A(\mathcal{P})) \subseteq \Psi(\Psi(\Gamma A(\mathcal{P})))$ holds. In order to show this relation we must show that for every argument $q[k] \in \mathcal{P}$ if $q[k] \in \Psi(\Gamma A(\mathcal{P}))$, then $q[k] \in \Psi(\Psi(\Gamma A(\mathcal{P}))$. Consider $q[k] \in \Psi(\Gamma A(\mathcal{P}))$. Then, $q[k]$ satisfies Definition 3.58 w.r.t. $A = \Gamma A(\mathcal{P})$. From comma one of this proof it follows that $\Gamma A(\mathcal{P}) \subseteq \Psi(\Gamma A(\mathcal{P}))$, consequently $q[k]$ satisfies Definition 3.58 w.r.t. $A = \Psi(\Gamma A(\mathcal{P}))$ too and so, $q[k] \in \Psi(\Psi(\Gamma A(\mathcal{P})))$.

Suppose that $\Psi^k(\Gamma A(\mathcal{P})) \subseteq \Psi^{k+1}(\Gamma A(\mathcal{P}))$ for $k > 0$. In order to show that $\Psi^{k+1}(\Gamma A(\mathcal{P})) \subseteq \Psi^{k+2}(\Gamma A(\mathcal{P}))$ we must show that for every argument $q[k] \in \mathcal{P}$ if $q[k] \in \Psi^{k+1}(\Gamma A(\mathcal{P}))$, then $q[k] \in \Psi^{k+2}(\Gamma A(\mathcal{P}))$. Consider $q[k] \in \Psi^{k+1}(\Gamma A(\mathcal{P}))$. Then $q[k]$ satisfies Definition 3.58 w.r.t. $A = \Psi^k(\Gamma A(\mathcal{P}))$. Since $\Psi^k(\Gamma A(\mathcal{P})) \subseteq \Psi^{k+1}(\Gamma A(\mathcal{P}))$, $q[k]$ satisfies Definition 3.58 w.r.t. $A = \Psi^{k+1}(\Gamma A(\mathcal{P}))$ too. Consequently, $q[k] \in \Psi^{k+2}(\Gamma A(\mathcal{P}))$. □

Observe that we can prove in a similar way that $AR(\mathcal{P}) \subseteq \Psi(AR(\mathcal{P}))$ and that $\Psi^i(AR(\mathcal{P})) \subseteq \Psi^{i+1}(AR(\mathcal{P}))$ for $i > 0$.

Definition 3.63 (Safe Arguments and Safe Programs). *For any program* $\mathcal{P}$*,* $safe(\mathcal{P}) = \Psi^{\infty}(\Gamma A(\mathcal{P}))$ *denotes the set of* safe arguments *of* $\mathcal{P}$*. A program* $\mathcal{P}$ *is said to be* safe *if all arguments are safe. The class of safe programs will be denoted by* $\mathcal{SP}$*.* □

Clearly, for any set of arguments $\mathcal{A} \subseteq \Gamma A(\mathcal{P})$, $\Psi^i(\mathcal{A}) \subseteq \Psi^i(\Gamma A(\mathcal{P}))$. Moreover, as shown in Proposition 3.62, when the starting set is $\Gamma A(\mathcal{P})$, the sequence $\Gamma A(\mathcal{P})$, $\Psi(\Gamma A(\mathcal{P})), \Psi^2(\Gamma A(\mathcal{P}))$, ... is monotone and there is a finite $n = O(|args(\mathcal{P})|)$ such that $\Psi^n(\Gamma A(\mathcal{P})) = \Psi^{\infty}(\Gamma A(\mathcal{P}))$. We can also define the inflactionary version of Ψ as $\hat{\Psi}(\mathcal{A}) = \mathcal{A} \cup \Psi(\mathcal{A})$, obtaining that $\hat{\Psi}^i(\Gamma A(\mathcal{P})) = \Psi^i(\Gamma A(\mathcal{P}))$, for all natural numbers i. The introduction of the inflactionary version guarantees that the sequence $\mathcal{A}$, $\hat{\Psi}(\mathcal{A})$, $\hat{\Psi}^2(\mathcal{A})$, ... is monotone for every set $\mathcal{A}$ of limited arguments. This would allow us to derive

a (possibly) larger set of limited arguments starting from any set of limited arguments.

Example 3.64. Consider again program $P_{3.53}$ of Example 3.53.
Although $AR(P_{3.53}) = \varnothing$, the program $P_{3.53}$ is safe as $\Sigma(P_{3.53})$ is acyclic.
Consider now the program $P_{3.59}$ of Example 3.59. As already shown in Example 3.59, the first application of the safety function to the set of Γ-acyclic arguments of $P_{3.59}$ gives $\Psi(\Gamma A(P_{3.59})) = \{\mathtt{b}[1], \mathtt{p}[1], \mathtt{p}[2]\}$. The application of the safety function to the obtained set gives $\Psi(\Psi(\Gamma A(P_{3.59}))) = \{\mathtt{b}[1], \mathtt{p}[1], \mathtt{p}[2], \mathtt{q}[1]\}$. In fact, in the unique rule defining `q`, term `f(Y)`, corresponding to the argument `q`[1], is limited in r w.r.t. $\{\mathtt{b}[1], \mathtt{p}[1], \mathtt{p}[2]\}$ (i.e. the variable `Y` appears in $body(r)$ in a term corresponding to argument `p`[2] and argument `p`[2], belonging to the input set, is limited). At this point, all arguments of $P_{3.59}$ belong to the resulting set. Thus, $safe(P_{3.59}) = args(P_{3.59})$, and we have that program $P_{3.59}$ is safe. □

We now show results on the expressivity of the class $\mathcal{SP}$ of safe programs.

Theorem 3.65. *The class $\mathcal{SP}$ of safe programs strictly includes the class $\Gamma\mathcal{A}$ of Γ-acyclic programs and is strictly contained in the class of terminating programs.*

Proof. ($\Gamma\mathcal{A} \subsetneq \mathcal{SP}$). From Proposition 3.62 it follows that $\Gamma\mathcal{A} \subseteq \mathcal{SP}$. Moreover, $\Gamma\mathcal{A} \subsetneq \mathcal{SP}$ as program $P_{3.59}$ is safe but not Γ-acyclic.

($\mathcal{SP} \subsetneq G_f$). From Proposition 3.60 it follows that every argument in the safe program can take values only from a finite domain. Consequently, the set of all possible derived ground terms is finite and the program is terminating. Moreover, we have that the program program:

$$
\begin{aligned}
&r_1 : \mathtt{p}(\mathtt{X}, \mathtt{X}) \leftarrow \mathtt{b}(\mathtt{X}).\\
&r_2 : \mathtt{q}(\mathtt{f}(\mathtt{X}), \mathtt{g}(\mathtt{X})) \leftarrow \mathtt{p}(\mathtt{X}, \mathtt{X}).\\
&r_3 : \mathtt{p}(\mathtt{X}, \mathtt{Y}) \leftarrow \mathtt{q}(\mathtt{X}, \mathtt{Y}).
\end{aligned}
$$

is terminating, but not safe. □

As a consequence of Theorem 3.65, every safe program admits a finite minimum model.

Complexity

We start by introducing a bound on the complexity of constructing the firing graph.

Proposition 3.66. *For any program $\mathcal{P}$, the firing graph $\Sigma(\mathcal{P})$ can be constructed in time $O(n_r^2 \times n_b \times (a_p \times a_f)^2)$, where n_r is the number of rules of $\mathcal{P}$, n_b is the maximum number of body atoms in a rule, a_p is the maximum arity of predicate symbols and a_f is the maximum arity of function symbols.*

Proof. To check whether a rule r_i fires a rule r_j we have to determine if an atom B in $body(r_j)$ unifies with the head-atom A of r_i. This can be done in time $O(n_b \times u)$, where u is the cost of deciding whether two atoms unify, which is quadratic in the size of the two atoms [78], that is $u = O((a_p \times a_f)^2)$ as the size of atoms is bounded by $a_p \times a_f$ (recall that the maximum depth of terms is 1). In order to construct the firing graph we have to consider all pairs of rules and for each pair we have to check if the first rule fires the second one. Therefore, the global complexity is $O(n_r^2 \times n_b \times u) = O(n_r^2 \times n_b \times (a_p \times a_f)^2)$. □

We recall that given two atoms A and B, the size of a m.g.u. θ for $\{A, B\}$ can be, in the worst case, exponential in the size of A and B, but the complexity of deciding whether a unifier for A and B exists is quadratic in the size of A and B [78].

Proposition 3.67. *The complexity of deciding whether a program $\mathcal{P}$ is safe is $O((size(\mathcal{P}))^2 + |args(\mathcal{P})|^3 \times |F_{\mathcal{P}}|)$.*

Proof. The construction of the firing graph $\Sigma(\mathcal{P})$ can be done in time $O(n_r^2 \times n_b \times (a_p \times a_f)^2)$, where n_r is the number of rules of $\mathcal{P}$, n_b is the maximum number of body atoms in a rule, a_p is the maximum arity of predicate symbols and a_f is the maximum arity of function symbols (cf. Proposition 3.66).

The complexity of computing $\Gamma A(\mathcal{P})$ is bounded by $O(|args(\mathcal{P})|^3 \times |F_{\mathcal{P}}|)$ (cf. Theorem 3.51).

From Definition 3.58 and Proposition 3.62 it follows that the sequence $\Gamma A(\mathcal{P}), \Psi(\Gamma A(\mathcal{P})), \Psi^2(\Gamma A(\mathcal{P}))$, ... is monotone and converges in a finite number of steps bounded by the cardinality of the set $args(\mathcal{P})$. The complexity of determining rules not depending on cycles in the firing graph $\Sigma(\mathcal{P})$ is bounded by $O(n_r^2)$, as it can be done by means of a depth-first traversal of $\Sigma(\mathcal{P})$, which is linear in the number of its edges. Since checking whether the conditions of Definition 3.57 hold for all arguments in $\mathcal{P}$ is in $O(size(\mathcal{P}))$, checking such conditions for at most $|args(\mathcal{P})|$ steps is $O(|args(\mathcal{P})| \times size(\mathcal{P}))$. Thus, the complexity of checking all the conditions of Definition 3.58 for all steps is $O(n_r^2 + |args(\mathcal{P})| \times size(\mathcal{P}))$.

Since, $n_r^2 \times n_b \times (a_p \times a_f)^2 = O((size(\mathcal{P}))^2)$, $|args(\mathcal{P})| = O(size(\mathcal{P}))$ and $n_r^2 = O((size(\mathcal{P}))^2)$, the complexity of deciding whether $\mathcal{P}$ is safe is $O((size(\mathcal{P}))^2 + |args(\mathcal{P})|^3 \times |F_{\mathcal{P}}|)$. □

3.6 Constraints-based approaches

This section will pinpoint the main limitations of the techniques presented so far, and will present a radically different approach to the problem of checking whether a logic program is limited by making use of linear constraints solving [16, 21]. In particular, the idea is to construct, for a given program $\mathcal{P}$ a set of linear constraints such that if there exists a solution to such set of constraints, then $\mathcal{P}$ is limited. This approach allows to identify many logic programs of practical use that are not identified as limited by previously presented techniques, like mapping-restriction, Γ-acyclic and safe programs.

3.6.1 Rule-bounded programs

We start by presenting *rule-bounded and cycle-bounded programs* [17], two classes of programs which are guaranteed to be limited and for which checking membership in the class is decidable.

Their definition relies on a novel technique which uses linear constraints to measure terms and atoms' sizes and checks if the size of the head of a rule is always bounded by the size of a mutually recursive body atom, which is not to be confused with mutually recursive predicate symbols (we will formally define what "mutually recursive" means in Definition 3.71 below).

Example 3.68. Consider the following program $\mathcal{P}_{3.68}$ implementing the bubble sort algorithm:

$$
\begin{array}{ll}
r_0: & \texttt{bub}(\texttt{L},[\,],[\,]) \leftarrow \texttt{input}(\texttt{L}).\\
r_1: & \texttt{bub}([\texttt{Y}|\texttt{T}],[\texttt{X}|\texttt{Cur}],\texttt{Sol}) \leftarrow \texttt{bub}([\texttt{X}|[\texttt{Y}|\texttt{T}]],\texttt{Cur},\texttt{Sol}), \texttt{X} \leq \texttt{Y}.\\
r_2: & \texttt{bub}([\texttt{X}|\texttt{T}],[\texttt{Y}|\texttt{Cur}],\texttt{Sol}) \leftarrow \texttt{bub}([\texttt{X}|[\texttt{Y}|\texttt{T}]],\texttt{Cur},\texttt{Sol}), \texttt{Y} < \texttt{X}.\\
r_3: & \texttt{bub}(\texttt{Cur},[\,],[\texttt{X}|\texttt{Sol}]) \leftarrow \texttt{bub}([\texttt{X}|[\,]],\texttt{Cur},\texttt{Sol}).
\end{array}
$$

where $[\cdot|\cdot]$ is a shortand for the list contruction function symbol $\texttt{lc}(\texttt{Head},\texttt{Tail})$. The list to be sorted is given by means of a fact of the form $\texttt{input}([\texttt{a}_1,...,\texttt{a}_\texttt{n}])$. The evaluation of this program always terminates for any input list. The ordered list $\texttt{Sol}$ can be obtained from the atom $\texttt{bub}([\,],[\,],\texttt{Sol})$ in the program's minimal model. □

Although the $\mathcal{P}_{3.68}$ is terminating, none of the termination criteria presented so far is able to realize it. One problem with them is that when they analyze how terms are propagated from the body to the head of rules, they look at arguments *individually*. For instance, in rule r_1 above, the simple fact that the second argument of `bub` has a size in the head greater than the one in the body prevents several techniques from realizing termination of the bottom-up evaluation of $\mathcal{P}_{3.68}$. More general classes such as mapping-restricted and bounded programs are able to do a more complex (yet limited) analysis of how some groups of arguments affect each other. Still, all current termination criteria are not able to realize that in every rule of $\mathcal{P}_{3.68}$ the *overall* size of the terms in the head does not increase w.r.t. the *overall* size of the terms in the body. One of the novelties of the technique proposed in this paper is the capability of doing this kind of analysis, thereby identifying programs (whose evaluation terminates) that none of the current techniques include.

The technique proposed in this paper easily realizes that the evaluation of $\mathcal{P}_{3.68}$ always terminates for any input list. In particular, this is done using linear constraints which measure the size of terms and atoms in order to check if the rules' head sizes are bounded by the size of some body atom when propagation occurs. Thus, this technique can understand that, in every rule, the overall size of the terms in the body does not increase during their propagation to the head, as there is only a simple redistribution of terms. Many practical programs dealing with lists and tree-like structures satisfy this property—below are two examples. However, this technique is not limited only to this kind of programs.

Example 3.69. Consider the program $\mathcal{P}_{3.69}$ below, performing a depth-first traversal of an input tree:

$$r_0: \ \mathtt{visit}(\mathtt{Tree}, [\,], [\,]) \leftarrow \mathtt{input}(\mathtt{Tree}).$$
$$r_1: \ \mathtt{visit}(\mathtt{Left}, [\mathtt{Root}|\mathtt{Visited}], [\mathtt{Right}|\mathtt{ToVisit}]) \leftarrow \mathtt{visit}(\mathtt{tree}(\mathtt{Root}, \mathtt{Left}, \mathtt{Right}), \mathtt{Visited}, \mathtt{ToVisit}).$$
$$r_2: \ \mathtt{visit}(\mathtt{Next}, \mathtt{Visited}, \mathtt{ToVisit}) \leftarrow \mathtt{visit}(\mathtt{null}, \mathtt{Visited}, [\mathtt{Next}|\mathtt{ToVisit}]).$$

The input tree is given with of facts like $\mathtt{input}(\mathtt{tree}(\mathtt{value}, \mathtt{left}, \mathtt{right}))$ where `tree` is a ternary function symbol used to represent tree structures. The program visits the nodes of the tree and puts them in a list following a depth-first search. The list `L` of visited elements can be obtained from the atom $\mathtt{visit}(\mathtt{null}, \mathtt{L}, [\,])$ in the program's minimal model. For instance, if the input tree is

$$\texttt{input(tree(a, tree(c, null, tree(d, null, null)), tree(b, null, null)))}.$$

the program produces the list $[\texttt{b}, \texttt{d}, \texttt{c}, \texttt{a}]$ containing the nodes of the tree in opposite order w.r.t. the traversal. □

Also in the case above, even if the program evaluation terminates for every input tree, none of the previously shown techniques is able to detect it.

Example 3.70. Finally, consider also the following program $\mathcal{P}_{3.70}$ computing the concatenation of two lists:

$$\begin{aligned}
r_0 &: \texttt{reverse}(\texttt{L}_1, [\,]) &&\leftarrow \texttt{input1}(\texttt{L}_1).\\
r_1 &: \texttt{reverse}(\texttt{L}_1, [\texttt{X}|\texttt{L}_2]) &&\leftarrow \texttt{reverse}([\texttt{X}|\texttt{L}_1], \texttt{L}_2).\\
r_2 &: \texttt{append}(\texttt{L}_1, \texttt{L}_2) &&\leftarrow \texttt{reverse}([\,], \texttt{L}_1),\ \texttt{input2}(\texttt{L}_2).\\
r_3 &: \texttt{append}(\texttt{L}_1, [\texttt{X}|\texttt{L}_2]) &&\leftarrow \texttt{append}([\texttt{X}|\texttt{L}_1], \texttt{L}_2).
\end{aligned}$$

Here `input1` and `input2` are used to store the lists $\texttt{L}_1$ and $\texttt{L}_2$ to be concatenated. The result list `L` can be retrieved from the atom $\texttt{append}([\,], \texttt{L})$ in the minimal model of $\mathcal{P}_{3.70}$. Clearly, the program is terminating. □

As already done with previous sections, we restrict our attention, w.l.o.g., to normal positive programs. Furthermore, we also assume that for any given program $\mathcal{P}$, no facts occur in $\mathcal{P}$. This assumption is w.l.o.g. as well, since rule-bounded programs are limited w.r.t. $\mathcal{B} = \mathit{pred}(\mathcal{P})$ (see Section 3.1). We also assume that no rules in a program $\mathcal{P}$ share logical variables and we assume that to each logical variable X occurring in $\mathcal{P}$, there corresponds a (unique) *integer variable* x (denoted by the same letter in lower case) which may occur in linear constraints.

In order to define the class of rule-bounded programs, we will make use of the firing graph introduced in Section 3.5. As an example, the firing graph $\Sigma(\mathcal{P}_{3.68})$ of program $\mathcal{P}_{3.68}$ of Example 3.68 is depicted in Figure 3.10.

We say that a rule r *depends on* a rule r' if r can be reached from r' through the edges of $\Sigma(\mathcal{P})$. A *strongly connected component* (SCC) of a directed graph G is a maximal set $\mathcal{C}$ of nodes of G s.t. every node of $\mathcal{C}$ can be reached from every node of $\mathcal{C}$ (through the edges in G). We say that an SCC $\mathcal{C}$ is *non-trivial* if there exists at least one edge in G between two not necessarily distinct nodes of $\mathcal{C}$. For instance, the firing graph in Figure 3.10 has two SCCs, $\mathcal{C}_1 = \{r_0\}$ and $\mathcal{C}_2 = \{r_1, r_2, r_3\}$, but only $\mathcal{C}_2$ is non-trivial. Given a program $\mathcal{P}$ and an SCC $\mathcal{C}$ of $\Sigma(\mathcal{P})$, $\mathit{pred}(\mathcal{C})$ denotes the set of predicate symbols defined by the

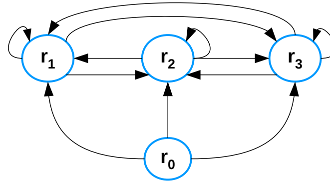


Fig. 3.10: Firing graph of $\mathcal{P}_{3.68}$.

rules in $\mathcal{C}$. We now define when the head atom and a body atom of a rule are mutually recursive.

Definition 3.71 (Mutually recursive atoms). *Let $\mathcal{P}$ be a program and r a rule in $\mathcal{P}$. The head atom $A = head(r)$ and an atom $B \in body(r)$ are* mutually recursive *if there is an SCC $\mathcal{C}$ of $\Sigma(\mathcal{P})$ s.t.:*

1. *$\mathcal{C}$ contains r, and*
2. *$\mathcal{C}$ contains a rule r' (possibly equal to r) s.t. $head(r')$ and B unify.* □

In the previous definition, when $r = r'$ we assume that r and r' are two "copies" that do not share any logical variable. Intuitively, the head atom A of a rule r and an atom B in the body of r are mutually recursive when there might be an actual propagation of terms from A to B (through the application of a sequence of rules). As a very simple example, if we have an SCC consisting only of the rule $\mathtt{p(f(X))} \leftarrow \mathtt{p(X)}, \mathtt{p(g(X))}$, the first body atom is mutually recursive with the head, while the second one is not as it does not unify with the head atom.

Given a rule r, we use $rbody(r)$ to denote the set of atoms in $body(r)$ which are mutually recursive with $head(r)$. Moreover, we define $sbody(r)$ as the set consisting of every atom in $body(r)$ that contains all logical variables appearing in $head(r)$, and define $srbody(r) = rbody(r) \cap sbody(r)$.

We say that a rule r in a program $\mathcal{P}$ is *relevant* if it is not a fact and the set of atoms $body(r) \setminus rbody(r)$ does not contain all logical variables in $head(r)$. Roughly speaking, a non-relevant rule will be ignored because either it cannot propagate terms or its head size is bounded by body atoms which are not mutually recursive with the head. We illustrate the notions introduced so far in the following example.

Example 3.72. Consider the following program $\mathcal{P}_{3.72}$:

$$r_1: \underbrace{\mathtt{s(f(X),Y)}}_{\mathrm{A}} \leftarrow \underbrace{\mathtt{q(X,f(Y))}}_{\mathrm{B}}, \underbrace{\mathtt{s(Z,f(Y))}}_{\mathrm{C}}.$$
$$r_2: \underbrace{\mathtt{q(f(U),V)}}_{\mathrm{D}} \leftarrow \underbrace{\mathtt{s(U,f(V))}}_{\mathrm{E}}.$$

The firing graph consists of the edges $\langle r_1, r_1\rangle$, $\langle r_1, r_2\rangle$, $\langle r_2, r_1\rangle$. Thus, there is only one SCC $\mathcal{C} = \{r_1, r_2\}$, which is non-trivial, and $pred(\mathcal{C}) = \{\mathtt{q}, \mathtt{s}\}$. Atoms A and B (resp. A and C, D and E) are mutually recursive. Moreover, $rbody(r_1) = \{B, C\}$, $srbody(r_1) = \{B\}$, $rbody(r_2) = srbody(r_2) = \{E\}$. Both r_1 and r_2 are relevant. □

We use $\mathbb{N}$ to denote the set of natural numbers $\{1, 2, 3, ...\}$ and $\mathbb{N}_0$ to denote the set of natural numbers including the zero. Moreover, $\mathbb{N}^k = \{(v_1, ..., v_k) \mid v_i \in \mathbb{N} \text{ for } 1 \leq i \leq k\}$ and $\mathbb{N}_0^k = \{(v_1, ..., v_k) \mid v_i \in \mathbb{N}_0 \text{ for } 1 \leq i \leq k\}$. Given a k-vector $\overline{v} = (v_1, ..., v_k)$ in $\mathbb{N}_0^k$, we use $\overline{v}[i]$ to refer to v_i, for $1 \leq i \leq k$. Given two k-vectors $\overline{v} = (v_1, ..., v_k)$ and $\overline{w} = (w_1, ..., w_k)$ in $\mathbb{N}_0^k$, we use $\overline{v} \cdot \overline{w}$ to denote the classical scalar product, i.e., $\overline{v} \cdot \overline{w} = \sum_{i=1}^{k} v_i \cdot w_i$.

As mentioned earlier, the basic idea of the proposed technique is to measure the size of terms and atoms in order to check if the rules' head sizes are bounded when propagation occurs. Thus, we introduce the notions of term and atom size.

Definition 3.73. *Let t be a term. The* size *of t is recursively defined as follows:*

$$size(t) = \begin{cases} x & \text{if } t \text{ is a logical variable } X; \\ m + \sum_{i=1}^{m} size(t_i) & \text{if } t = f(t_1, ..., t_m). \end{cases}$$

where x is an integer variable. The size *of an atom $A = p(t_1, ..., p_n)$, denoted $size(A)$, is the n-vector $(size(t_1), ..., size(t_n))$.* □

In the definition above, an integer variable x intuitively represents the possible sizes that the logical variable X can have during the evaluation. The size of a term of the form $f(t_1, ..., t_m)$ is defined by summing up the size of its terms t_i's plus the arity m of f. Note that from the definition above, the size of every constant is 0.

Example 3.74. Consider rule r_1 of program $\mathcal{P}_{3.68}$ (see Example 3.68). Using `lc` to denote the list constructor operator "|", the rule can be rewritten as follows:

$$\mathtt{bub(lc(Y,T),lc(X,Cur),Sol)} \leftarrow \mathtt{bub(lc(X,lc(Y,T)),Cur,Sol)}, \mathtt{X} \leq \mathtt{Y}.$$

Let A (resp. B) be the atom in the head (resp. the first atom in the body). Then,

$$size(A) = (2 + y + t, \quad 2 + x + cur, \quad sol)$$
$$size(B) = (2 + [x + (2 + y + t)], \quad cur, \quad sol)$$

□

We are now ready to define rule-bounded programs.

Definition 3.75 (Rule-bounded programs). *Let $\mathcal{P}$ be a program, $\mathcal{C}$ a non-trivial SCC of $\Sigma(\mathcal{P})$, and $pred(\mathcal{C}) = \{p_1, ..., p_k\}$. We say that $\mathcal{C}$ is* rule-bounded *if there exist k vectors $\overline{\alpha}_{p_h} \in \mathbb{N}^{arity(p_h)}$, $1 \leq h \leq k$, such that for every relevant rule $r \in \mathcal{C}$ with $A = head(r) = p_i(t_1, ..., t_n)$, there exists an atom $B = p_j(u_1, ..., u_m)$ in $srbody(r)$ s.t. the following inequality is satisfied*

$$\overline{\alpha}_{p_j} \cdot size(B) - \overline{\alpha}_{p_i} \cdot size(A) \geq 0$$

for every non-negative value of the integer variables in $size(B)$ and $size(A)$.

We say that $\mathcal{P}$ is rule-bounded *if every non-trivial SCC of $\Sigma(\mathcal{P})$ is rule-bounded.* □

Intuitively, for every relevant rule of a non-trivial SCC of $\Sigma(\mathcal{P})$, Definition 3.75 checks if the size of the head atom is bounded by the size of a mutually recursive body atom for all possible sizes the terms can assume.

Example 3.76. Consider again program $\mathcal{P}_{3.72}$ of Example 3.72. Recall that the only non-trivial SCC of $\Sigma(\mathcal{P}_{3.72})$ is $\mathcal{C} = \{r_1, r_2\}$, and both r_1 and r_2 are relevant. To determine if the program is rule-bounded we need to check if $\mathcal{C}$ is rule-bounded. Thus, we need to find $\overline{\alpha}_q, \overline{\alpha}_s \in \mathbb{N}^2$ such that there is an atom in *srbody*(r_1) and an atom in *srbody*(r_2) which satisfy the two inequalities derived from r_1 and r_2 for all non-negative values of the integer variables therein. Since both *srbody*(r_1) and *srbody*(r_2) contain only one element, we have only one choice, namely the one where B is selected for r_1 and E is selected for r_2.
Thus, we need to check if there exist $\overline{\alpha}_q, \overline{\alpha}_s \in \mathbb{N}^2$ s.t. the following linear constraints are satisfied for all non-negative values of the integer variables appearing in them:

$$\begin{cases} \overline{\alpha}_q \cdot size(B) - \overline{\alpha}_s \cdot size(A) \geq 0 \\ \overline{\alpha}_s \cdot size(E) - \overline{\alpha}_q \cdot size(D) \geq 0 \end{cases} \Rightarrow \begin{cases} \overline{\alpha}_q \cdot (x, 1 + y) - \overline{\alpha}_s \cdot (1 + x, y) \geq 0 \\ \overline{\alpha}_s \cdot (u, 1 + v) - \overline{\alpha}_q \cdot (1 + u, v) \geq 0 \end{cases}$$

By expanding the scalar products and isolating every integer variable we obtain:

$$\begin{cases} (\overline{\alpha}_q[1] - \overline{\alpha}_s[1]) \cdot x + (\overline{\alpha}_q[2] - \overline{\alpha}_s[2]) \cdot y + (\overline{\alpha}_q[2] - \overline{\alpha}_s[1]) \geq 0 \\ (\overline{\alpha}_s[1] - \overline{\alpha}_q[1]) \cdot u + (\overline{\alpha}_s[2] - \overline{\alpha}_q[2]) \cdot v + (\overline{\alpha}_s[2] - \overline{\alpha}_q[1]) \geq 0 \end{cases}$$

The previous inequalities must hold for all $x, y, u, v \in \mathbb{N}_0$; it is easy to see that this is the case iff the following system admits a solution:

$$\begin{cases} \overline{\alpha}_q[1] - \overline{\alpha}_s[1] \geq 0, \quad \overline{\alpha}_q[2] - \overline{\alpha}_s[2] \geq 0, \quad \overline{\alpha}_q[2] - \overline{\alpha}_s[1] \geq 0, \\ \overline{\alpha}_s[1] - \overline{\alpha}_q[1] \geq 0, \quad \overline{\alpha}_s[2] - \overline{\alpha}_q[2] \geq 0, \quad \overline{\alpha}_s[2] - \overline{\alpha}_q[1] \geq 0 \end{cases}$$

Since a solution does exist, e.g. $\overline{\alpha}_s[1] = \overline{\alpha}_s[2] = \overline{\alpha}_q[1] = \overline{\alpha}_q[2] = 1$ (recall that every $\overline{\alpha}[i]$ must be greater than 0), the SCC $\mathcal{C}$ is rule-bounded, and so is the program. □

The method in the previous example to find vectors $\overline{\alpha}_p$ for all $p \in pred(\mathcal{C})$ can always be applied. That is, we can always isolate the integer variables in the original inequalities and then derive one inequality for each expression that multiplies an integer variable plus the one for the constant term, imposing that all such expressions must be greater than or equal to 0—we precisely state this property in Lemma 3.99.

It is worth noting that the proposed technique can easily recognize many terminating practical programs where terms are simply exchanged from the body to the head of rules (e.g., see Examples 3.68, 3.69, and 3.70).

Example 3.77. Consider program $\mathcal{P}_{3.68}$ of Example 3.68. Recall that the only non-trivial SCC of $\Sigma(\mathcal{P}_{3.68})$ is $\{r_1, r_2, r_3\}$ (see Figure 3.10) and all rules in it are relevant. Since $|srbody(r_i)| = 1$ for every r_i in the SCC, we have only one set of inequalities, which is the following one after isolating integer variables(we assume that the empty list is represented by a simple constant):

$$\begin{cases} (\overline{\alpha}_b[1] - \overline{\alpha}_b[2]) \cdot x_1 + (2\overline{\alpha}_b[1] - 2\overline{\alpha}_b[2]) \geq 0 \\ (\overline{\alpha}_b[1] - \overline{\alpha}_b[2]) \cdot y_2 + (2\overline{\alpha}_b[1] - 2\overline{\alpha}_b[2]) \geq 0 \\ (\overline{\alpha}_b[1] - \overline{\alpha}_b[3]) \cdot x_3 + (\overline{\alpha}_b[2] - \overline{\alpha}_b[1]) \cdot cur_3 + (2\overline{\alpha}_b[1] - 2\overline{\alpha}_b[3]) \geq 0 \end{cases}$$

where subscript b stands for predicate symbol `bub`, whereas subscripts associated with integer variables are used to refer to the occurrences of logical variables in different rules (e.g., y_2 is the integer variable associated to the

logical variable `Y` in rule r_2). A possible solution is $\overline{\alpha}_b = (1, 1, 1)$ and thus $\mathcal{P}_{3.68}$ is rule-bounded.

Considering program $\mathcal{P}_{3.69}$ of Example 3.69, we obtain the following constraints:

$$\begin{cases} (\overline{\alpha}_v[1] - \overline{\alpha}_v[2]) \cdot root_1 + (\overline{\alpha}_v[1] - \overline{\alpha}_v[3]) \cdot right_1 + (3\overline{\alpha}_v[1] - 2\overline{\alpha}_v[2] - 2\overline{\alpha}_v[3]) \geq 0 \\ (\overline{\alpha}_v[3] - \overline{\alpha}_v[1]) \cdot next_2 + 2\overline{\alpha}_v[3] \geq 0 \end{cases}$$

where subscript v stands for predicate symbol `visit`. By setting $\overline{\alpha}_v = (2, 1, 2)$, we get positive integer values of $\overline{\alpha}_v[1], \overline{\alpha}_v[2], \overline{\alpha}_v[3]$ s.t. the inequalities above are satisfied for all $root_1$, $right_1$, $next_2 \in \mathbb{N}_0$. Thus, $\mathcal{P}_{3.69}$ is rule-bounded.

The firing graph of program $\mathcal{P}_{3.70}$ of Example 3.70 has two non-trivial SCCs $\mathcal{C}_1 = \{r_1\}$ and $\mathcal{C}_2 = \{r_3\}$. The constraints for $\mathcal{C}_1$ are:

$$\begin{cases} (\overline{\alpha}_r[1] - \overline{\alpha}_r[2]) \cdot x_1 + (2\overline{\alpha}_r[1] - 2\overline{\alpha}_r[2]) \geq 0 \end{cases}$$

where subscript r stands for predicate symbol `reverse`. It is easy to see that by choosing any (positive integer) values of $\overline{\alpha}_r[1]$ and $\overline{\alpha}_r[2]$ such that $\overline{\alpha}_r[1] \geq \overline{\alpha}_r[2]$, the inequality above holds for all $x_1 \in \mathbb{N}_0$. Likewise, the constraints for $\mathcal{C}_2$ are

$$\begin{cases} (\overline{\alpha}_a[1] - \overline{\alpha}_a[2]) \cdot x_3 + (2\overline{\alpha}_a[1] - 2\overline{\alpha}_a[2]) \geq 0 \end{cases}$$

where subscript a stands for predicate symbol `append`. By choosing any (positive integer) values of $\overline{\alpha}_a[1]$ and $\overline{\alpha}_a[2]$ such that $\overline{\alpha}_a[1] \geq \overline{\alpha}_a[2]$, the inequality above holds for all $x_3 \in \mathbb{N}_0$. Thus, $\mathcal{P}_{3.70}$ is rule-bounded. □

Correctness and expressiveness

In this section, we show that every rule-bounded program is terminating and provide results on the relative expressiveness of rule-bounded programs and other criteria.

Note that every program $\mathcal{P}$ can be partitioned into an ordered sequence of sub-programs $\mathcal{P}_1, ..., \mathcal{P}_n$, called *stratification*, such that, for every $1 \leq i \leq n$, every rule r in $\mathcal{P}_i$ depends only on rules belonging to some sub-program $\mathcal{P}_j$ with $1 \leq j \leq i$. Recall that a rule r depends on a rule r' if r can be reached from r' through the edges of the firing graph. Moreover, there always exists a stratification where every sub-program $\mathcal{P}_i$ is either a non-trivial SCC or a set of trivial SCCs. Given a set of facts D, it is well known that $\mathcal{MM}(\mathcal{P} \cup D)$ can be defined in terms of the minimal model of the $\mathcal{P}_i$'s following the order of

the partition as follows: if $M_0 = D$ and $M_i = \mathcal{MM}(\mathcal{P}_i \cup M_{i-1})$ for $1 \leq i \leq n$, then $M_n = \mathcal{MM}(\mathcal{P} \cup D)$.

Lemma 3.78. *A program $\mathcal{P}$ is terminating iff every non-trivial SCC of $\Sigma(\mathcal{P})$ is terminating.*

Proof. ($\Rightarrow$) Clearly, if there is an SCC which is not terminating, then $\mathcal{P}$ is not terminating.
($\Leftarrow$) Assume now that $\mathcal{P}$ does not terminate and all its non-trivial SCCs terminates. This means that there is a set of facts D such that the fixpoint of $\mathcal{P} \cup D$ is not finite. Since $\mathcal{P} \cup D$ can be partitioned into $(\mathcal{P}_1, ..., \mathcal{P}_n)$, there must be a non-trivial (i.e. recursive) SCC $\mathcal{P}_i$ such that $\mathcal{P}_i \cup M_{i-1}$ does not terminate. This contradicts the hypothesis that all non-trivial SCCs terminate. Indeed if P_i terminates, then for every set of facts D' including the facts in M_{i-1}, the fixpoint of $\mathcal{P}_i \cup D'$ terminates and, therefore, the fixpoint of $\mathcal{P}_i \cup M_{i-1}$ terminates as well. □

We now refine the previous lemma by showing that to see if a program $\mathcal{P}$ is terminating it is not necessary to analyze every non-trivial SCC entirely, but we can focus on its relevant rules. Henceforth, for every set of rules $\mathcal{C}$, we use $Rel(\mathcal{C})$ to denote the set of relevant rules of $\mathcal{C}$.

Lemma 3.79. *Let $\mathcal{P}$ be a program and let $\mathcal{C}$ be an SCC of $\Sigma(\mathcal{P})$. Then, $\mathcal{C}$ is terminating iff $Rel(\mathcal{C})$ is terminating.*

Proof. It follows from the fact that we can derive only a finite number of ground atoms using the rules in $ground(\mathcal{C}) \setminus ground(Rel(\mathcal{C}))$ starting from a finite set of facts—recall that, by definition, every non-relevant rule has a set of atoms in the body that are not mutually recursive with the head and contain all variables in the head. □

To show the correctness of our approach, we first show that every rule-bounded program can be rewritten into an "equivalent" program belonging to a simpler class of programs, called *unary-size-restricted*. Then, we prove that unary-size-restricted programs are terminating and this entails that rule-bounded programs are terminating as well.

Definition 3.80 (Program expansion). *Let $\mathcal{P}$ be a program and let $\omega = \{\overline{\omega}_{p_1}, ..., \overline{\omega}_{p_n}\}$ be a set of vectors such that $\overline{\omega}_{p_i} \in \mathbb{N}^{arity(p_i)}$ and $p_i \in pred(\mathcal{P})$ for $1 \leq i \leq n$. For any atom $A = p(t_1, ..., t_m)$ occurring in $\mathcal{P}$, we define $ex(A, \omega) = A$, if $p \notin pred(\mathcal{P})$, otherwise $ex(A, \omega) = p(\overline{t}_1, ..., \overline{t}_m)$, where each*

$\bar{t}_j$ is the sequence $t_j, ..., t_j$ of length $\omega_p[j]$. Finally, $\mathcal{P}^\omega$ denotes the program derived from $\mathcal{P}$ by replacing every atom A with $ex(A, \omega)$. □

Intuitively, the expansion of a program is obtained from the original program by increasing the arity of each predicate symbol according to ω. Below is an example.

Example 3.81. Consider program $P_{3.72}$ of Example 3.72 and the set of vectors $\omega = \{\overline{\omega}_\mathtt{s}, \overline{\omega}_\mathtt{q}\}$ where $\overline{\omega}_\mathtt{s} = (2, 3)$ and $\overline{\omega}_\mathtt{q} = (2, 1)$. The program $ex(P_{3.72}, \omega)$ is as follows:

$$\begin{array}{ll} r_1: & \mathtt{s(f(X), f(X), Y, Y, Y)} \leftarrow \mathtt{q(X, X, f(Y))}, \ \ \mathtt{s(Z, Z, f(Y), f(Y), f(Y))}. \\ r_2: & \mathtt{q(f(U), f(U), V)} \qquad \leftarrow \mathtt{s(U, U, f(V), f(V), f(V))}. \end{array}$$

□

We now show that for every program $\mathcal{P}$ and every set of vectors ω, $\mathcal{P}$ is terminating iff $ex(\mathcal{P}, \omega)$ is terminating. In the following, for every program $\mathcal{P}$, we define $\omega(\mathcal{P}) = \{ \ \{\overline{\omega}_{p_1}, ..., \overline{\omega}_{p_n}\} \mid p_i \in pred(\mathcal{P}) \wedge \overline{\omega}_{p_i} \in \mathbb{N}^{arity(p_i)}\}$.

Lemma 3.82. *For every program $\mathcal{P}$ and every $\omega \in \omega(\mathcal{P})$, $\mathcal{P}$ is terminating iff $ex(\mathcal{P}, \omega)$ is terminating.*

Proof. For every atom A^ω occurring in $ex(\mathcal{P}, \omega)$ let A be the corresponding atom in $\mathcal{P}$. The claim follows from the observation that whenever there is a instance D such that $T^\infty_{\mathcal{P} \cup D}(\varnothing)$ is infinite, it is always possible to construct the instance $ex(D, \omega)$ which guarantees that $T^\infty_{ex(\mathcal{P},\omega) \cup ex(D,\omega)}(\varnothing)$ is infinite as well.
Conversely, for every instance D^ω of $ex(\mathcal{P}, \omega)$, if $T^\infty_{ex(\mathcal{P},\omega) \cup D^\omega}(\varnothing)$ is infinite, then we can always construct the instance D guaranteeing that $T^\infty_{\mathcal{P} \cup D}(\varnothing)$ is infinite as well. □

We now introduce the class of unary-size-restricted programs and show that such programs are terminating. To this aim, we define the *total size* of an atom $A = p(t_1, ..., t_n)$ as $tsize(A) = \sum_{i=1}^{n} size(t_i)$.

Definition 3.83 (Unary-size-restricted program). *A program $\mathcal{P}$ is said to be* unary-size-restricted *if for every rule $r \in \mathcal{P}$ which is not a fact, there is an atom B in $sbody(r)$ such that $tsize(B) \geq tsize(head(r))$ for every non-negative value of the integer variables occurring in $tsize(B)$ and $tsize(head(r))$.* □

Theorem 3.84. *Every unary-size-restricted program is terminating.*

Proof. Let $\mathcal{P}$ be a unary-size-restricted program and D a finite set of facts, we consider only rules in $\mathcal{P}$ having a non-empty body. Given an atom A and a ground instance A' of A, let θ be the mgu of A and A'. Notice that θ is of the form $\{X_1/t_1, ..., X_n/t_n\}$ where the X_i's are exactly the logical variables occurring in A and all the t_j's are ground terms. It can be easily verified that $tsize(A')$ can be obtained from $tsize(A)$ by replacing every integer variable x_i in $tsize(A)$ with $size(t_i)$.

We now show that for every ground rule $r' \in ground(\mathcal{P})$ there is an atom $B' \in body(r')$ such that $tsize(B') \geq tsize(head(r'))$. Consider a rule r in $\mathcal{P}$ of the form $A \leftarrow B_1, ..., B_k$ and a ground rule $r' \in ground(r)$ of the form $A' \leftarrow B'_1, ..., B'_k$. Since $\mathcal{P}$ is unary-size-restricted, there exists an atom B_j in $sbody(r)$ such that $tsize(B_j) \geq tsize(A)$ for every non-negative value of the integer variables occurring in the inequality. Notice every logical variable in A appears also in B_j by definition of *sbody*. Let $\{X_1/t_1, ..., X_n/t_n\}$ be the mgu of B_j and B'_j. As $tsize(B_j) \geq tsize(A)$ holds for all non-negative value of its integer variables, it also holds when every integer variable x_i is replaced with $size(t_i)$, for $1 \leq i \leq n$. Thus, $tsize(B'_j) \geq tsize(A')$.

Let us denote $T^i_{\mathcal{P}\cup D}(\varnothing)$ as M_i for every $i \geq 1$ and let $tsize_{max} = \max\{tsize(B) \mid B \leftarrow$ is a fact in $\mathcal{P} \cup D\}$. We show that for every $i \geq 1$ and every ground atom A in M_i the following holds $tsize_{max} \geq tsize(A)$. The proof is by induction on i.

- *Base case (i=1).* It follows from the fact that $M_1 = \{B \mid B \leftarrow$ is a fact in $\mathcal{P} \cup D\}$.
- *Inductive step ($i \to i+1$).* Let r' be a ground rule in $ground(\mathcal{P})$ such that $body(r') \subseteq M_i$. Then, as shown above, there is an atom B in $body(r')$ such that $tsize(B) \geq tsize(head(r'))$. By the induction hypothesis, $tsize_{max} \geq tsize(B)$ and thus $tsize_{max} \geq tsize(head(r'))$.

Thus, for every $i \geq 1$ and every ground atom A in M_i, we have that $tsize(A)$ is bounded by $tsize_{max}$. Since programs are range-restricted, atoms in $\cup_{i\geq 1} M_i$ are built from constants and function symbols appearing in $\mathcal{P} \cup D$, which are finitely many. These observations and the definition of *tsize* imply that we can have only finitely many ground atoms in $\cup_{i\geq 1} M_i$. Hence, $\mathcal{P}$ is terminating. □

We are now ready to show the correctness of the rule-bounded technique.

Theorem 3.85. *Every rule-bounded program is terminating.*

Proof. Let $\mathcal{P}$ be a rule-bounded program and $\mathcal{C}$ a non-trivial SCC of $\Sigma(\mathcal{P})$. Since $\mathcal{P}$ is rule-bounded, then there exists $\omega \in \omega(\mathcal{C})$ which satisfies the condition of Definition 3.75, that is, $\mathcal{C}$ is rule-bounded. This implies that $Rel(\mathcal{C})^\omega$ is unary-size-restricted. Thus, $Rel(\mathcal{C})^\omega$ is terminating by Theorem 3.121. Lemma 3.118 implies that $Rel(\mathcal{C})$ is terminating and Lemma 3.79 in turn implies that $\mathcal{C}$ is terminating. Finally, by Lemma 3.78, we can conclude that $\mathcal{P}$ is terminating. □

The class of rule-bounded programs is incomparable with different termination criteria in the literature, including the most general ones.

Theorem 3.86. *Rule-bounded programs are incomparable with argument-restricted, mapping-restricted, and bounded programs.*

Proof. Recall that both bounded and mapping-restricted programs include argument-restricted programs. To prove the claim we show that *(i)* there is a program which is rule-bounded but is neither mapping-restricted nor bounded, and *(ii)* there is a program which is argument-restricted but not rule-bounded. *(i)* As already shown, program $\mathcal{P}_{3.68}$ of Example 3.68 is rule-bounded; however, it can be easily verified that $\mathcal{P}_{3.68}$ is neither mapping-restricted nor bounded. *(ii)* Consider the program consisting of the rules $\mathtt{p(f(X))} \leftarrow \mathtt{q(X)}$ and $\mathtt{q(Y)} \leftarrow \mathtt{p(f(Y))}$. This program is argument-restricted (and thus also mapping-restricted and bounded) but is not rule-bounded. □

Regarding the termination criteria mentioned in Theorem 3.86, we recall that mapping restriction (MR) and bounded programs (BP) are incomparable and both extend argument restriction (AR). Concerning the computational complexity, while AR is polynomial time, both MR and BP are exponential. As a remark, it is interesting to note that the above result highlights the fact that our technique analyzes logic programs from a radically different point of view w.r.t. previously defined approaches, which analyze how complex terms are propagated among arguments.

3.6.2 Cycle-bounded Programs

As saw in the previous section, to determine if a program is rule-bounded we check through linear constraints if the size of the head atom is bounded by the size of a body atom for every relevant rule in a non-trivial SCC of the firing graph (cf. Definition 3.75). Looking at each rule individually has its limitations, as shown by the following example.

Example 3.87. Consider the following simple program $\mathcal{P}_{3.87}$:

$$
\begin{aligned}
r_1 :\ & \mathtt{p(X,Y)} && \leftarrow \mathtt{q(f(X),Y)}.\\
r_2 :\ & \mathtt{q(W,f(Z))} && \leftarrow \mathtt{p(W,Z)}.
\end{aligned}
$$

It is easy to see that the program above is terminating, but it is not rule-bounded. The linear inequalities for the program are (cf. Definition 3.75):

$$
\begin{cases}
(\overline{\alpha}_q[1] - \overline{\alpha}_p[1]) \cdot x + (\overline{\alpha}_q[2] - \overline{\alpha}_p[2]) \cdot y + \overline{\alpha}_q[1] \geq 0\\
(\overline{\alpha}_p[1] - \overline{\alpha}_q[1]) \cdot w + (\overline{\alpha}_p[2] - \overline{\alpha}_q[2]) \cdot z - \overline{\alpha}_q[2] \geq 0
\end{cases}
$$

It can be easily verified that there are no positive integer values for $\overline{\alpha}_p[1]$, $\overline{\alpha}_p[2]$, $\overline{\alpha}_q[1]$, $\overline{\alpha}_q[2]$ such that the inequalities hold for all $x, y, w, z \in \mathbb{N}_0$. The reason is the presence of the expression $-\overline{\alpha}_q[2]$ in the second inequality. Intuitively, this is because the size of the head atom increases w.r.t. the size of the body atom in r_2. However, notice that the cycle involving r_1 and r_2 does not increase the overall size of propagated terms. This suggests we can check if an *entire cycle* (rather than each individual rule) propagates terms of bounded size. □

To deal with programs like the one shown in the previous example, we introduce the class of *cycle-bounded programs*, which is able to perform an analysis of how terms propagate through a *group* of rules, rather than looking at rules *individually* as done by the rule-bounded criterion.

Given a program $\mathcal{P}$, a cyclic path π of $\Sigma(\mathcal{P})$ is a sequence of edges $\langle r_1, r_2\rangle, \langle r_2, r_3\rangle, ..., \langle r_n, r_1\rangle$. Moreover a cyclic path π is *basic* if every edge π does not occur more than once. We say that π is *relevant* if every r_i is relevant, for $1 \leq i \leq n$.

In the following, we first present the cycle-bounded criterion for linear programs and then show how it can be applied to non-linear ones.

Dealing with linear programs. A program $\mathcal{P}$ is *linear* if every rule in $\mathcal{P}$ is linear. A rule r is *linear* if $|rbody(r)| \leq 1$. Notice that $rbody(r)$ contains exactly one atom B for every linear rule r in a non-trivial SCC of the firing graph; thus, with a slight abuse of notation, we use $rbody(r)$ to refer to B.

Definition 3.88 (Cycle constraints). *Let $\mathcal{P}$ be a linear program and let $\pi = \langle r_1, r_2\rangle, ..., \langle r_n, r_1\rangle$ be a basic cyclic path of $\Sigma(\mathcal{P})$. For every mgu θ_i of $head(r_i)$ and $rbody(r_{i+1})$ $(1 \leq i < n)$*[2]*, we define the set of (linear) equalities $eq(\theta_i) = \{x = size(t) \mid X/t \in \theta_i\}$. Then, we define $eq(\pi) = \bigcup_{1 \leq i < n} eq(\theta_i)$.* □

[2] Note that such θ_i's always exist by definition of firing graph.

Example 3.89. Consider the program $\mathcal{P}_{3.87}$ and the two basic cyclic paths $\pi_1 = \langle r_1, r_2\rangle\,\langle r_2, r_1\rangle$ and $\pi_2 = \langle r_2, r_1\rangle\,\langle r_1, r_2\rangle$ of $\Sigma(\mathcal{P}_{3.87})$. The mgu of $head(r_1)$ and $rbody(r_2)$ is $\theta = \{\mathtt{X/W},\ \mathtt{Y/Z}\}$ and thus $eq(\pi_1) = \{x = w,\ y = z\}$. Furthermore, the mgu of $head(r_2)$ and $rbody(r_1)$ is $\theta = \{\mathtt{W/f(X)},\ \mathtt{Y/f(Z)}\}$ and thus $eq(\pi_2) = \{w = 1 + x,\ y = 1 + z\}$. □

Definition 3.90 (Linear cycle-bounded programs). *Let $\mathcal{P}$ be a linear program, $\pi = \langle r_1, r_2\rangle \dots \langle r_n, r_1\rangle$ be a basic cyclic path of $\Sigma(\mathcal{P})$ and p be the predicate defined by r_n. We say that π is* cycle-bounded *if $eq(\pi)$ is satisfiable for some non-negative value of its integer variables and there exists a vector $\overline{\alpha}_p \in \mathbb{N}^{arity(p)}$ such that the constraint*

$$\overline{\alpha}_p \cdot size(rbody(r_1)) - \overline{\alpha}_p \cdot size(head(r_n)) \geq 0$$

is satisfied for every non-negative value of its integer variables that satisfy $eq(\pi)$. We say that $\mathcal{P}$ is cycle-bounded *if every relevant basic cyclic path of $\Sigma(\mathcal{P})$ is cycle-bounded.* □

In the definition above, we assume that distinct basic cyclic paths do not share any logical variable.

Example 3.91. Consider again program $\mathcal{P}_{3.87}$ of Example 3.87. The program is clearly linear and $\Sigma(\mathcal{P}_{3.87})$ has only two relevant basic cyclic paths $\pi_1 = \langle r_1, r_2\rangle\langle r_2, r_1\rangle$ and $\pi_2 = \langle r_2, r_1\rangle\langle r_1, r_2\rangle$. To check if $\mathcal{P}_{3.87}$ is cycle-bounded, we need to check if $eq(\pi_1) = \{x_1 = w_1,\ y_1 = z_1\}$ and $eq(\pi_2) = \{w_2 = 1 + x_2,\ y_2 = 1 + z_2\}$ admit a solution and if there exist $\overline{\alpha}_p, \overline{\alpha}_q \in \mathbb{N}^2$ s.t. the constraints:

$$\begin{aligned} \overline{\alpha}_q \cdot (x_1 + 1, y_1) &- \overline{\alpha}_q \cdot (w_1, z_1 + 1) \geq 0, \\ \overline{\alpha}_p \cdot (w_2, z_2) &- \overline{\alpha}_p \cdot (x_2, y_2) \geq 0 \end{aligned}$$

are satisfied for all $x_1, y_1, w_1, z_1 \in \mathbb{N}_0$ and all $x_2, y_2, w_2, z_2 \in \mathbb{N}_0$ that satisfy $eq(\pi_1)$ and $eq(\pi_2)$.

By applying the equality conditions $eq(\pi_1)$ and $eq(\pi_2)$ to the above constraints we get the below inequalities for the basic cyclic paths π_1 and π_2:

$$\begin{aligned} (\overline{\alpha}_q[1], \overline{\alpha}_q[2]) \cdot (x_1 + 1, z_1) &- (\overline{\alpha}_q[1], \overline{\alpha}_q[2])) \cdot (x_1, z_1 + 1) \geq 0, \\ (\overline{\alpha}_p[1], \overline{\alpha}_p[2])) \cdot (x_2 + 1, z_2) &- (\overline{\alpha}_p[1], \overline{\alpha}_p[2])) \cdot (x_2, z_2 + 1) \geq 0 \end{aligned}$$

It is easy to see that the first constraint (resp. the second) is satisfied for every vector $\overline{\alpha}_p \in \mathbb{N}^2$ (resp. $\overline{\alpha}_q \in \mathbb{N}^2$) such that $\overline{\alpha}_p[1] \geq \overline{\alpha}_p[2]$ (resp. $\overline{\alpha}_q[1] \geq \overline{\alpha}_q[2]$). Thus, $\mathcal{P}_{3.87}$ is cycle-bounded. □

To prove the correctness of our approach, we introduce a simpler class of terminating programs, as we did in the case of rule-bounded programs.

Definition 3.92 (Linear cycle-size-bounded programs). *Let $\mathcal{P}$ be a linear program. We say that $\mathcal{P}$ is* cycle-size-bounded *if for every relevant basic cyclic path $\pi = \langle r_1, r_2 \rangle \ldots \langle r_n, r_1 \rangle$ of $\Sigma(\mathcal{P})$, $eq(\pi)$ is satisfiable for some non-negative value of its integer variables and the constraint*

$$tsize(rbody(r_1)) - tsize(head(r_n)) \geq 0$$

is satisfied for every non-negative value of its integer variables that satisfy $eq(\pi)$. □

Theorem 3.93. *Every linear cycle-size-bounded program is terminating.*

Proof. Let $\mathcal{P}$ be a cycle-size-bounded program and D a finite set of facts. Consider a relevant basic cyclic path $\pi = \langle r_1, r_2 \rangle \ldots \langle r_n, r_1 \rangle$ of $\Sigma(\mathcal{P})$. Let $r'_1, \ldots, r'_n$ be ground rules s.t. $r'_i \in ground(r_i)$ for $1 \leq i \leq n$ and $head(r'_i) = rbody(r'_{i+1})$ for $1 \leq i < n$. For $1 \leq i \leq n$, let θ^h_i be the mgu of $head(r_i)$ and $head(r'_i)$, and θ^b_i the mgu of $rbody(r_i)$ and $rbody(r'_i)$. Then,

- $tsize(head(r'_i))$ can be obtained from $tsize(head(r_i))$ by replacing every integer variable x in $tsize(head(r_i))$ with $size(t)$ provided that $X/t \in \theta^h_i$, for $1 \leq i \leq n$;
- $tsize(rbody(r'_i))$ can be obtained from $tsize(rbody(r_i))$ by replacing every integer variable x in $tsize(rbody(r_i))$ with $size(t)$ provided that $X/t \in \theta^b_i$, for $1 \leq i \leq n$;
- if we replace every integer variable x in $eq(\pi)$ with $size(t)$ iff X/t belongs to $\cup^n_{i=1}(\theta^h_i \cup \theta^b_i)$, then $eq(\pi)$ is satisfied.

The items above entail that $tsize(rbody(r'_1)) - tsize(head(r'_n)) \geq 0$. This means that we cannot derive atoms of increasing size through the cyclic application of rules and thus $\mathcal{P} \cup D$ is terminating. □

Theorem 3.94 (Soundness). *Every linear cycle-bounded program is terminating.*

Proof. The proof is similar to the one presented for rule-bounded programs. Given a linear cycle-bounded program $\mathcal{P}$, we are going to construct an equivalent program (like $ex(\mathcal{P}, \omega)$) to $\mathcal{P}$ as follows: for every relevant basic cyclic path $\pi = \langle r_1, r_2 \rangle \ldots \langle r_n, r_1 \rangle$ of $\Sigma(\mathcal{P})$, let $\overline{\alpha}_p$ be the vector such that $\overline{\alpha}_p \cdot size(rbody(r_1)) - \overline{\alpha}_p \cdot size(head(r_n)) \geq 0$. Then, remove rules

r_1 and r_n from $\mathcal{P}$ and insert the rules $head(r_1) \leftarrow ex(rbody(r_1), \overline{\alpha}_p)$ and $ex(head(r_n), \overline{\alpha}_p) \leftarrow rbody(r_n)$ respectively. Finally, in order to preserve the firing of rules in the obtained program, for every pair of basic cyclic paths $\pi_1 = \langle r_1, r_2\rangle \ldots \langle r_n, r_1\rangle$, $\pi_2 = \langle s_1, s_2\rangle \ldots \langle s_m, s_1\rangle$, where p is the predicate defined by r_n and s_n with arity k, add to $\mathcal{P}$ a rule of the form $ex(A, \overline{\alpha}_p) \leftarrow ex(A, \overline{\beta}_p)$, where A is the atom $p(X_1, ..., X_k)$ and $\overline{\alpha}_p$, $\overline{\beta}_p$ are the vectors such that $\overline{\alpha}_p \cdot size(rbody(r_1)) - \overline{\alpha}_p \cdot size(head(r_n)) \geq 0$ and $\overline{\beta}_p \cdot size(rbody(s_1)) - \overline{\beta}_p \cdot size(head(s_m)) \geq 0$ respectively. It is not difficult to show that the obtained program is terminating iff $\mathcal{P}$ is terminating. Moreover, since $\mathcal{P}$ is cycle-bounded the new program is consequently cycle-size-bounded. From Theorem 3.93, we get that the new program is terminating and so it is $\mathcal{P}$. □

Dealing with non-linear programs. The application of the cycle-bounded criterion to arbitrary programs consists in applying the technique to a set of linear programs derived from the original one. Given a rule r, the set of *linear versions* of r is defined as the set of rules $\ell(r) = \{head(r) \leftarrow B \mid B \in rbody(r)\}$. Given a program $\mathcal{P} = \{r_1, ..., r_n\}$, the set of *linear versions* of $\mathcal{P}$ is defined as the set of linear programs $\ell(\mathcal{P}) = \{\{r'_1, ..., r'_n\} \mid r'_i \in \ell(r_i)$ for $1 \leq i \leq n\}$.

Definition 3.95 (Cycle-bounded programs). *A (possibly non-linear) program $\mathcal{P}$ is* cycle-bounded *if every (linear) program in $\ell(\mathcal{P})$ is cycle-bounded.* □

Theorem 3.96. *Every cycle-bounded program is terminating.*

Proof. Notice that every linear version $\mathcal{P}' \in \ell(\mathcal{P})$ of $\mathcal{P}$ is such that for every set of facts D, $\mathcal{MM}(D \cup \mathcal{P}) \subseteq \mathcal{MM}(D \cup \mathcal{P}')$. Thus, if every linear version of $\mathcal{P}$ is cycle-bounded then for every set of facts D, $\mathcal{MM}(D \cup \mathcal{P})$ is finite. □

Theorem 3.97 (Expressivity). *Cycle-bounded programs are incomparable with rule-bounded, argument-restricted, mapping-restricted and bounded programs.*

Proof. As shown in Example 3.87, program $\mathcal{P}_{3.87}$ is cycle-bounded, but it can be easily verified that it is neither mapping-restricted (and thus not argument-restricted) nor rule-bounded. Moreover, the one rule program $\{\texttt{p(X, Y, f(Z, W))} \leftarrow \texttt{p(f(Z, Y), X, W)}.\}$ is cycle-bounded but it is not bounded. Conversely, the program $\{\texttt{p(f(X))} \leftarrow \texttt{p(f(f(X))), p(X)}.\}$ is rule-bounded, argument-restricted (and thus mapping-restricted) and bounded but not cycle-bounded. □

3.6.3 Complexity

In this section, we provide upper bounds for the time complexity of checking whether a program is rule-bounded or cycle-bounded.

Lemma 3.98. *Given a program $\mathcal{P}$, constructing $\Sigma(\mathcal{P})$ is in PTIME.*

Proof. The construction of $\Sigma(\mathcal{P})$ requires checking, for every atom A in the head of a rule and every atom B in the body of a rule, if A and B unify. Since we need to check $|\mathcal{P}| \times \sum_{r \in \mathcal{P}} |body(r)|$ times if two atoms unify and checking whether two atoms A and B unify can be done in quadratic time w.r.t. $||A||$ and $||B||$ [78], then the construction of $\Sigma(\mathcal{P})$ is in $PTIME$. □

It is worth noting that the number of SCCs is bounded by $O(|\mathcal{P}|)$ and that after having built $\Sigma(\mathcal{P})$, the cost of checking whether a SSC is trivial or nontrivial is constant, whereas the cost of checking whether a rule is relevant is bounded by $O(||\mathcal{P}||)$. Inequalities associated with basic cycles can be rewritten by grouping terms with respect to integer coefficients (also called α-coefficients) or with respect to integer variables. Therefore, in the following we assume that inequalities grouped with respect to integer variables are of the form $\gamma_1 \cdot x_1, + \cdots + \gamma_n \cdot x_n + \gamma_0 \geq 0$, where each γ_i, for $0 \leq i \leq n$, is an arithmetic expression built by using α-coefficients and natural numbers, whereas inequalities grouped with respect to integer coefficients are of the form $\alpha_1 \cdot w_1, + \cdots + \alpha_m \cdot w_m \geq 0$, where each w_j, for $1 \leq i \leq m$, is an arithmetic expression built by using integer variables and natural numbers. Obviously, each γ_i can be considered an integer coefficient, whereas each w_j can be considered an integer variable.

Lemma 3.99. *Consider a linear inequality of the form*

$$\gamma_1 \cdot x_1 + \ldots + \gamma_n \cdot x_n + \gamma_0 \geq 0$$

where the γ_i's are integer coefficients and the x_j's are integer variables. The inequality is satisfied for every non-negative value of the x_j's iff $\gamma_i \geq 0$ for every $0 \leq i \leq n$.

Proof. ($\Leftarrow$) Straightforward. ($\Rightarrow$) By contradiction, assume that the inequality is satisfied for every non-negative value of the integer variables occurring in it, but there exists $0 \leq i \leq n$ such that $\gamma_i < 0$. If $1 \leq i \leq n$, then the inequality is not satisfied when $x_i = \lfloor abs(\gamma_{n+1}/\gamma_i) \rfloor + 1$ and $x_j = 0$ for every $j \neq i$. If $i = 0$, then the inequality is not satisfied when $x_j = 0$ for every $1 \leq j \leq n$. □

Theorem 3.100. *Checking whether a program $\mathcal{P}$ is rule-bounded is in NP.*

Proof. In order to check whether $\mathcal{P}$ is rule-bounded we need to: 1) construct the firing graph $\Sigma(\mathcal{P})$ of $\mathcal{P}$, 2) compute the SCCs of $\Sigma(\mathcal{P})$, and 3) check if every non-trivial SCC is rule-bounded.
1) The construction of the firing graph is in *PTIME* by Lemma 3.98.
2) It is well known that computing the SCCs of a directed graph can be done in linear time w.r.t. the number of nodes and edges. Since the number of nodes of $\Sigma(\mathcal{P})$ is $|\mathcal{P}|$ and the maximum number of edges of $\Sigma(\mathcal{P})$ is $|\mathcal{P}|^2$, then computing all the SCCs is clearly in *PTIME*.
3) Let $\mathcal{C}$ be a non-trivial SCC of $\Sigma(\mathcal{P})$, $n = O(|\mathcal{P}|)$ the number of relevant rules in $\mathcal{C}$, v the maximum number of distinct variables occurring in the head atoms of the relevant rules in $\mathcal{C}$, and a the maximum arity of the predicate symbols in $pred(\mathcal{C})$. Since it is always possible to rewrite the constraints as in Definition 3.75 in the form presented by Lemma 3.99, given a fixed choice of one atom in $srbody(r)$ for every relevant rule r of $\mathcal{C}$, checking whether $\mathcal{C}$ is rule-bounded according to that choice can be done by solving a set of at most $n \times (v + 1)$ linear constraints with at most $2 \times a$ non-negative coefficients per constraint—clearly, the size of the set of constraints is bounded by $O(||\mathcal{P}||)$ and if the set of constraints admit a solution, then there is a solution where the size of the α-coefficients is polynomial in the size of $||\mathcal{P}||$ (bounded by $O(v \times n \times k)$, where k is the maximum constant appearing in the set of inequalities). As checking if such a set of linear constraints admits a solution can be done in non-deterministic polynomial time [70], it follows from the above discussion that this can be checked in polynomial time.

Hence, checking whether $\mathcal{P}$ is rule-bounded is in *NP*. □

We discuss now the complexity of checking whether a program is cycle-bounded. To this aim, we first introduce a technical lemma similar to Lemma 3.99.

Lemma 3.101. *Consider a linear inequality of the form*

$$\alpha_1 \cdot w_1 + \ldots + \alpha_n \cdot w_n < 0 \tag{3.2}$$

where the w_i's are integer variables and the α_j's positive integer coefficients. The inequality is satisfied iff $w_i \leq 0$ for every $1 \leq i \leq n$ and $w_j < 0$ for some $1 \leq j \leq n$.

Proof. ($\Leftarrow$) It follows straightforwardly from the fact that each $\alpha_j > 0$ for every $j \in [1,n]$.
($\Rightarrow$) By contradiction, assume that (3.2) is satisfied for every $\alpha_j > 0$, where $j \in [1,n]$, but either there is $i \in [1,n]$ such that $w_i > 0$ or $w_i \leq 0$ for every $i \in [1,n]$ but none of such inequalities is strict. If there is $i \in [1,n]$, ($i = 1$, for example) such that $w_1 > 0$, then, since $\alpha_j > 0$ for each $j \in [1,n]$, any assignment of $\alpha_1, ..., \alpha_n > 0$ such that $\alpha_1 > |\alpha_2 \cdot w_2 + ... + \alpha_n \cdot w_n|$ will not satisfy (3.2). In the case whether no $w_i \leq 0$ is strict, then $w_i = 0$ for every $i \in [1,n]$ and thus $\alpha_1 \cdot w_1 + ... + \alpha_n \cdot w_n$ will be zero, which does not satisfy (3.2). □

The next result says that checking if a program $\mathcal{P}$ is cycle-bounded is in *coNP*. We recall that a given a set of linear constraints depending on some integer variables is satisfiable if there exist non-negative integer values of its integer variables that satisfy the constraints. A solution of such linear constraints is any assignment for their integer variables to some non-negative integer values satisfying the constraints.

Theorem 3.102. *Checking whether a program $\mathcal{P}$ is cycle-bounded is in coNP.*

Proof. In order to prove the claim, we focus on the complement of our problem. By definition, a program $\mathcal{P}$ is not cycle-bounded if there exists a linear version $\mathcal{P}'$ of $\mathcal{P}$ which is not cycle-bounded, which means that a relevant basic cyclic path $\pi = \langle r_1, r_2\rangle ... \langle r_n, r_1\rangle$ of $\Sigma(\mathcal{P}')$ is such that either $eq(\pi)$ is not satisfiable or there is a solution of $eq(\pi)$ for which the inequality $\overline{\alpha}_p \cdot size(rbody(r_1)) - \overline{\alpha}_p \cdot size(head(r_n)) \geq 0$ is false, for every $\overline{\alpha}_p \in \mathbb{N}^{arity(p)}$. Checking the statement above can be carried out by the following non-deterministic procedure.

Guess a linear version $\mathcal{P}'$ of $\mathcal{P}$ and a basic cyclic path π of $\Sigma(\mathcal{P}')$ and check If π is relevant. if it is not, then reject (i.e., the program is cycle-bounded). Then, check if $eq(\pi)$ is satisfiable, if it is not then accept (i.e., the program is not cycle-bounded). Now, it remains to check whether there is a solution of $eq(\pi)$ such that $\overline{\alpha}_p \cdot size(rbody(r_1)) - \overline{\alpha}_p \cdot size(head(r_n)) \geq 0$ is false for all $\overline{\alpha}_p \in \mathbb{N}^{arity(p)}$. To accomplish the aforementioned task, we can check wheher $\overline{\alpha}_p \cdot size(rbody(r_1)) - \overline{\alpha}_p \cdot size(head(r_n)) < 0$ is true. Moreover, isolating every term $\overline{\alpha}_p[i]$ ($1 \leq i \leq arity(p)$) in the inequality, we get an expression of the form $\overline{\alpha}_p[1] \cdot w_1 + ... + \overline{\alpha}_p[arity(p)] \cdot w_{arity(p)} < 0$, where each w_i depends only on variables occurring in $eq(\pi)$. Since from Lemma 3.101, this is equivalent

to check whether $w_i \leq 0$ for $i \in [1,n]$ and there is $j \in [1,n]$ such that $w_j < 0$, checking whether there is a solution of $eq(\pi)$ such that $\overline{\alpha}_p \cdot size(rbody(r_1)) - \overline{\alpha}_p \cdot size(head(r_n)) \geq 0$ is false for all $\overline{\alpha}_p \in \mathbb{N}^{arity(p)}$ is equivalent to guessing a $j \in [1,n]$ and check that the set of linear constraints $eq(\pi) \cup \{w_1 \leq 0\} \cup \cdots \cup \{w_j < 0\} \cup \cdots \cup \{w_n \leq 0\}$ is satisfiable. The input program is not cycle-bounded iff the previous set of linear constraints is satisfiable.

To show the desired upper bound, note that guessing a linear version $\mathcal{P}'$ of $\mathcal{P}$ and a basic cyclic path of $\Sigma(\mathcal{P}')$ can be done in non-deterministic polynomial time, since $|\mathcal{P}'| = |\mathcal{P}|$ and the maximum length of a basic cyclic path coincides with the number of edges of $\Sigma(\mathcal{P}')$. Moreover, as previously stated, constructing the firing graph is feasible in deterministic polynomial time. Furthermore, the construction of $eq(\pi)$ can be carried on in polynomial time too, by using a polynomially sized representation of the mgu's of the rules occurring in π [78]. Finally, as shown in [70], checking whether the set of linear constraints $eq(\pi) \cup \{w_1 \leq 0\} \cup \cdots \cup \{w_j < 0\} \cup \cdots \cup \{w_n \leq 0\}$ is satisfiable is in NP. □

3.6.4 Extending rule boundedness: Size-restriction

In this section we will point out the limitations of the rule-bounded technique by means of examples and then introduce a refined version of the approach, giving rise to the class of size-restricted programs [21], which is a wider class of practical logic programs. Also in this case, we restrict our attention to normal positive logic programs.

Example 3.103. Consider the following program $\mathcal{P}_{4.51}$:

$$\mathtt{p(f(X,X),Y,Z)} \leftarrow \mathtt{p(X,g(Z),g(Y))}.$$

The program evaluation always terminates whatever finite set of facts is added to the program—however, this program is neither rule-bounded nor cycle-bounded. □

The reason why the rule-bounded (or cycle-bounded) criterion is not able to identify the program above as terminating is because of the fact that it analyzes atoms in a whole, thus not understanding that the growth of the term in $\mathtt{p}[1]$ is bounded by the shrinking of terms in $\mathtt{p}[2], \mathtt{p}[3]$. For the opposite reason, previous techniques which analyze programs only at the argument level, cannot understand the relation between $\mathtt{p}[1]$ and the arguments $\mathtt{p}[2], \mathtt{p}[3]$.

To provide a practical example, below we report a general program which recognizes strings of the language corresponding to an arbitrary $LR(1)$ grammar.

Example 3.104. Consider the following program $\mathcal{P}_{3.104}$:

$$
\begin{array}{lll}
\texttt{par}(\texttt{T}, [\texttt{S1}|[\texttt{Sym}|[\texttt{St}|\texttt{L}]]]) & \leftarrow & \texttt{par}([\texttt{Sym}|\texttt{T}], [\texttt{St}|\texttt{L}]), \\
 & & \texttt{act}(\texttt{St}, \texttt{Sym}, \texttt{shift}(\texttt{S1})). \\
\texttt{red}([\texttt{Sym}|\texttt{T}], [\texttt{St}|\texttt{L}], \texttt{A}, \texttt{B}) & \leftarrow & \texttt{par}([\texttt{Sym}|\texttt{T}], [\texttt{St}|\texttt{L}]), \\
 & & \texttt{act}(\texttt{St}, \texttt{Sym}, \texttt{reduce}(\texttt{A}, \texttt{B})). \\
\texttt{red}(\texttt{I}, \texttt{L}, \texttt{A}, \texttt{T}) & \leftarrow & \texttt{red}(\texttt{I}, [\texttt{S}|[\texttt{X}|\texttt{L}]], \texttt{A}, [\texttt{Y}|\texttt{T}]). \\
\texttt{par}(\texttt{I}, [\texttt{S1}|[\texttt{A}|[\texttt{St}|\texttt{L}]]]) & \leftarrow & \texttt{red}(\texttt{I}, [\texttt{St}|\texttt{L}], \texttt{A}, []), \\
 & & \texttt{act}(\texttt{St}, \texttt{A}, \texttt{goto}(\texttt{S1})).
\end{array}
$$

where we use the classical syntax [H|T] for a list. $LR(1)$ grammars can be encoded in a standard form using an action table defined by facts of the form act(⟨state⟩, ⟨symbol⟩, ⟨operation⟩).

Specifically, given the current parsing state ⟨state⟩ and a symbol ⟨symbol⟩ to be parsed, ⟨operation⟩ describes one of the following four parsing operations: shift(⟨newstate⟩), i.e., the next token is read from the input and pushed to the parsing stack along with the new parsing state ⟨newstate⟩; reduce(A, B), i.e., there is a production rule $\texttt{A} \rightarrow \texttt{B}$ in the grammar and the top of the parsing stack contains B (according to ⟨state⟩), which must be replaced with A; goto(⟨newstate⟩), i.e., once the reduce operation is complete, the parsing state changes accordingly; accept, i.e., the input string is accepted. The computation starts by providing as input the action table and a fact of the form $\texttt{par}([\texttt{a}_1, ..., \texttt{a}_\texttt{n}, \$], [\texttt{s}_0])$, where $[\texttt{a}_1, ..., \texttt{a}_\texttt{n}, \$]$ is the input string, followed by the "end of string symbol" \$, and $[\texttt{s}_0]$ is the parsing stack containing the initial state $\texttt{s}_0$. The string is accepted iff the program model contains two atoms of the form par([\$], [s|L]) and act(s, \$, accept). □

Once again, the program above terminates for every finite set of facts. While none of the previously presented approaches is able to realize it, the new technique detects the program as terminating.

To overcome the limitations presented above, we will adopt an hybrid approach where arguments can be grouped in an arbitrary way, in order to find possible relations between them. At the same time, this will allow us to drop the all-or-nothing approach of rule-bounded and cycle-bounded programs (either we can say that the program is terminating or we cannot say

anything), and identify arguments that are "limited" even when the program is not entirely recognized as terminating. Furthermore, as we will see next, the technique will also be able to leverage external information about limited arguments.

We start by generalizing the notion of relevant rule, seen in Section 3.6.1. Given a program $\mathcal{P}$ and a set $\mathcal{A}$ of limited arguments of $\mathcal{P}$, we say that a rule $r \in \mathcal{P}$ is $\mathcal{A}$-*relevant* if $head(r)$ contains at least one variable which does not appear in $body(r) \setminus rbody(r)$ and does not appear in a term t_i of a body atom $p(t_1, ..., t_n)$ such that $p[i] \in \mathcal{A}$. Rules that are not $\mathcal{A}$-relevant will not be considered in the analysis of an SCC (Definition 3.107) because they cannot infinitely propagate terms (when the SCC is considered in isolation), as all head variables appear in either a body atom which is not mutually recursive with the head or in correspondence of a limited argument. The following example illustrates the aforementioned notions.

Example 3.105. Consider the following program $\mathcal{P}_{3.105}$:

$$r_1: \underbrace{\mathtt{p(f(X),Y)}}_{\mathtt{A}} \leftarrow \underbrace{\mathtt{p(X,f(Y))}}_{\mathtt{B}}, \underbrace{\mathtt{b(X,Z)}}_{\mathtt{C}}.$$
$$r_2: \underbrace{\mathtt{p(X,g(Y))}}_{\mathtt{D}} \leftarrow \underbrace{\mathtt{p(f(X),Y)}}_{\mathtt{E}}.$$

Given the set of limited arguments $\mathcal{A} = \{\mathtt{p}[2]\}$, rule r_2 is $\mathcal{A}$-relevant, since variable X occurring in D appears only in the mutually recursive body atom E inside argument $\mathtt{p}[1]$, which is not in $\mathcal{A}$. Conversely, r_1 is not $\mathcal{A}$-relevant, since variables X, Y appearing in A occur in the body respectively in the non-mutually recursive atom C and inside $\mathtt{p}[2] \in \mathcal{A}$ of atom B. □

As mentioned before, one of the features of this technique is the capability of leveraging information about arguments that are known to be limited. In order to enable the technique to exploit this kind of information, several notions introduced in the following are defined w.r.t. a set $\mathcal{A}$ of arguments, to be read as the set of arguments that are known to be limited when our criterion is applied to a given program.

Definition 3.106 (Argument/predicate domain). *Given a program $\mathcal{P}$ and a set of arguments $\mathcal{A}$, the* domain *of an argument $p[i] \in args(\mathcal{P})$ w.r.t. $\mathcal{A}$, denoted $\mathbb{D}_{\mathcal{A}}(p[i])$, is $\mathbb{Z}$ if $p[i] \in \mathcal{A}$, and $\mathbb{N}$ otherwise. The* domain *of a predicate symbol p of arity n is $\mathbb{D}_{\mathcal{A}}(p) = \mathbb{D}_{\mathcal{A}}(p[1]) \times \cdots \times \mathbb{D}_{\mathcal{A}}(p[n])$.* □

Below we define when an argument is $\mathcal{A}$-*size-restricted* in an SCC of a program—as shown in the following, this ensures that the argument is limited when the SCC is considered in isolation. Then, in Definition 3.112, we will define how to combine the information coming from all the SCCs in order to determine whether or not an argument is $\mathcal{A}$-size-restricted in the entire program.

Definition 3.107 (Size-restricted arguments in an SCC). *Consider a program $\mathcal{P}$ and a set $\mathcal{A}$ of limited arguments of $\mathcal{P}$. Let $\mathcal{C}$ be an SCC of $\mathcal{P}$ with $pred(\mathcal{C}) = \{p_1, ..., p_n\}$. We say that an argument $p_i[j]$ of $\mathcal{C}$ is* $\mathcal{A}$-size-restricted in $\mathcal{C}$ *iff*

1. *for every rule $r \in \mathcal{C}$ such that $head(r) = p_i(t_1, ..., t_m)$ the following condition holds: for every variable X occurring in t_j, there exists a term u_k of a body atom $q(u_1, ..., u_{m'})$ s.t. X occurs in u_k and $q[k] \in \mathcal{A}$; or*
2. *there exist n vectors $\overline{\alpha}_h \in \mathbb{D}_{\mathcal{A}}(p_h)$, $1 \leq h \leq n$, such that for every $\mathcal{A}$-relevant rule $r \in \mathcal{C}$ there exists an atom B in $body(r)$ such that if $pr(head(r)) = p_k$ and $pr(B) = p_l$, then the following conditions hold:*
   a) *the constraint*
   $$\overline{\alpha}_l \cdot size(B) \geq \overline{\alpha}_k \cdot size(head(r))$$
   is satisfied for every non-negative value of the integer variables in it; and
   b) *if $p_k = p_i$ then either $\overline{\alpha}_i[j] \neq 0$ or the constraint*
   $$\overline{\alpha}_l \cdot size(B) > \overline{\alpha}_i \cdot size(head(r))$$
   is satisfied for every non-negative value of the integer variables in it. □

Condition 1 of the definition above simply checks if $p_i[j]$ is $\mathcal{A}$-size-restricted because for every rule of $\mathcal{C}$ having p_i in the head, all variables appearing in correspondence of $p_i[j]$ appear in the body in correspondence of a limited argument.

As for Condition 2, roughly speaking, Definition 3.107 says that an argument $p_i[j]$ is $\mathcal{A}$-size-restricted in an SCC if, for every (relevant) rule, the size of part of the head is always bounded by the size of part of a body atom, to within a constant factor. When $\overline{\alpha}_i[j] = 0$, a stricter inequality must be satisfied for the rules having p_i in the head. When other coefficients are 0, we are considering only parts of atoms in the analysis—e.g., assuming that $\overline{\alpha}_k[1] = 0$, this means that the first term in every p_k-atom is ignored in the

analysis. Notice that only the coefficients associated with limited arguments can assume arbitrary values in $\mathbb{Z}$. We notice that while the rule-bounded criterion allows positive coefficients only, here we allow coefficients to be zero and take negative values (this last case applies to limited arguments only).

Example 3.108. Consider program $\mathcal{P}_{4.51}$ of Example 4.51, reported below:

$$\mathtt{p(f(X,X),Y,Z)} \leftarrow \mathtt{p(X,g(Z),g(Y))}.$$

Let us consider $\mathcal{A} = \varnothing$. The program has only one SCC $\mathcal{C}$ consisting of the rule above, which is $\mathcal{A}$-relevant. The vector $\overline{\alpha}_p = (0,1,1)$ allows us to say that all arguments are $\mathcal{A}$-size-restricted in $\mathcal{C}$. In fact, when arguments $\mathtt{p[2]}$ and $\mathtt{p[3]}$ are considered, Condition 2(a) of Definition 3.107 holds since

$$(0,1,1)\cdot(x,z+1,y+1) \geq (0,1,1)\cdot(2x+2,y,z)$$

is satisfied for all non-negative values of the integer variables, and Condition 2(b) is trivially satisfied because both $\overline{\alpha}_p[2]$ and $\overline{\alpha}_p[3]$ are not 0. When argument $\mathtt{p[1]}$ is considered, Condition 2(a) is the same as before and thus is satisfied, and Condition 2(b) holds too since the constraint above with a strict inequality is still satisfied for all non-negative values of the integer variables. □

Example 3.109. Consider again program $\mathcal{P}_{3.104}$ of Example 3.104, which has only one SCC $\mathcal{C}$ coinciding with $\mathcal{P}_{3.104}$ itself. Let us consider $\mathcal{A} = \varnothing$. All rules are $\mathcal{A}$-relevant. We now show that every argument is $\mathcal{A}$-size-restricted in $\mathcal{C}$. In particular, consider the inequalities associated with the rules of $\mathcal{P}_{3.104}$ when the $\mathtt{act}$-atoms are selected in the body of the first, second, and fourth rule, and the $\mathtt{red}$-atom is selected for the third rule:

$$\begin{cases} \overline{\alpha}_{act}\cdot(st,sym,1{+}s_1) \geq \overline{\alpha}_{par}\cdot(t,6{+}s_1{+}sym{+}st{+}l) \\ \overline{\alpha}_{act}\cdot(st,sym,2{+}a{+}b) \geq \overline{\alpha}_{red}\cdot(2{+}sym{+}t,2{+}st{+}l,a,b) \\ \overline{\alpha}_{red}\cdot(i,4{+}s{+}x{+}l,a,2{+}y{+}t) \geq \overline{\alpha}_{red}\cdot(i,l,a,t) \\ \overline{\alpha}_{act}\cdot(st,a,1{+}s_1) \geq \overline{\alpha}_{par}\cdot(i,6{+}s_1{+}a{+}st{+}l) \end{cases}$$

By incorporating the vectors $\overline{\alpha}_{act} = (1,1,1)$, $\overline{\alpha}_{par} = (0,0)$, $\overline{\alpha}_{red} = (0,0,1,1)$ into the constraints above, we obtain:

$$\begin{cases} st + sym + s_1 + 1 & \geq \ 0 \\ st + sym + a + b + 2 & \geq \ a + b \\ a + y + t + 2 & \geq \ a + t \\ st + a + s_1 + 1 & \geq \ 0 \end{cases}$$

It is easy to see that the constraints above are satisfied for every st, sym, s_1, a, b, y, $t \in \mathbb{N}$, and thus Condition 2(a) of Definition 3.107 holds for all arguments. Moreover, since $\overline{\alpha}_{act}[1]$, $\overline{\alpha}_{act}[2]$, $\overline{\alpha}_{act}[3]$, $\overline{\alpha}_{red}[3]$, and $\overline{\alpha}_{red}[4]$ are all different from 0, we can say that arguments $\mathtt{act}[1], \mathtt{act}[2], \mathtt{act}[3], \mathtt{red}[3]$, and $\mathtt{red}[4]$ are $\mathcal{A}$-size-restricted in $\mathcal{C}$, as Condition 2(b) is also satisfied. For arguments $\mathtt{par}[1], \mathtt{par}[2], \mathtt{red}[1]$, and $\mathtt{red}[2]$ (whose coefficients are 0), we have to check if the constraints associated with the rules having predicate symbol `par` (resp. `red`) in the head, namely the first and the last one (resp. the second and third one), are satisfied with a strict inequality. As this is the case, Condition 2(b) holds, and arguments $\mathtt{par}[1]$, $\mathtt{par}[2], \mathtt{red}[1]$, and $\mathtt{red}[2]$ are $\mathcal{A}$-size-restricted in $\mathcal{C}$. □

We now define how to determine if an argument is $\mathcal{A}$-size-restricted in the entire program. This is done by combining the information obtained from the individual analysis of the SCCs. We start by introducing some additional notions.

Given a program $\mathcal{P}$, we assume an arbitrary but fixed numbering $\mathcal{C}_1, ..., \mathcal{C}_n$ of its SCCs. We also define *ex-args*$(\mathcal{P})$ as the set $\{p[i/j] \mid \mathcal{C}_j$ is an SCC of $\mathcal{P}$ and $p[i] \in$ *args*$(\mathcal{C}_j)\}$. Each element of *ex-args*$(\mathcal{P})$ is called an *extended argument* of $\mathcal{P}$. The next tool is called *extended argument graph*—a directed graph keeping track of the propagation of terms between arguments. It is a refinement of the argument graph of [26] and it leverages the firing graph to perform a component-wise analysis of how terms are propagated between arguments and to get rid of propagation (between arguments) that cannot really occur.

Definition 3.110 (Extended argument graph). *The* extended argument graph *of a program $\mathcal{P}$, denoted $\Delta(\mathcal{P})$, is a directed graph whose set of nodes is ex-args$(\mathcal{P})$ and where there is an edge $\langle q[j/k], p[i/l]\rangle$ iff*

- *$k = l$ and there is a rule $r \in \mathcal{C}_k$ such that (1) head(r) is a p-atom, (2) there is a q-atom B in body(r), (3) the i-th term of head(r) and j-th term of B have a common variable, and (4) there is a rule $r' \in \mathcal{P}$ such that head(r') and B unify; or*
- *$k \neq l$ and $p = q$, $i = j$, and there are two rules $r_1 \in \mathcal{C}_k$ and $r_2 \in \mathcal{C}_l$ such that pr(head$(r_1)) = p$ and $\langle r_1, r_2\rangle$ is an edge of $\Sigma(\mathcal{P})$.* □

Intuitively, an edge $\langle q[j/k], p[i/l]\rangle$ of $\Delta(\mathcal{P})$ means that there can be a propagation of terms from $q[j]$ in component $\mathcal{C}_k$ to $p[i]$ in component $\mathcal{C}_l$. We

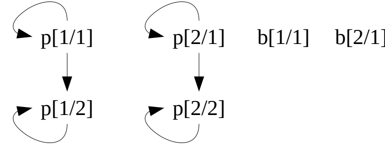


Fig. 3.11: Extended argument graph of $\mathcal{P}_{3.105}$.

say that an extended argument $p[i/l]$ *depends on* an extended argument $q[j/k]$ if there is a path from the latter to the former in $\Delta(\mathcal{P})$.

Example 3.111. Consider again program $\mathcal{P}_{3.105}$ of Example 3.105. Figure 3.11 illustrates $\Delta(\mathcal{P}_{3.105})$. □

We are now ready to define when an argument is $\mathcal{A}$-size-restricted in a program.

Definition 3.112 ($\mathcal{A}$-size-restricted arguments/programs). *Let $\mathcal{P}$ be a program and $\mathcal{A}$ be a set of limited arguments of $\mathcal{P}$. An argument $p[i]$ is $\mathcal{A}$-size-restricted in $\mathcal{P}$ if for every SCC $\mathcal{C}_l$ of $\mathcal{P}$ such that $p \in pred(\mathcal{C}_l)$,*

1. *$p[i]$ is $\mathcal{A}$-size-restricted in $\mathcal{C}_l$, and*
2. *$p[i/l]$ depends only on extended arguments $q[j/k]$ such that $q[j]$ is $\mathcal{A}$-size-restricted in $\mathcal{C}_k$.*

We denote by $\mathcal{R}_{\mathcal{A}}(\mathcal{P})$ the set of all $\mathcal{A}$-size-restricted arguments in $\mathcal{P}$. We say that $\mathcal{P}$ is $\mathcal{A}$-size-restricted *iff $args(\mathcal{P}) = \mathcal{A} \cup \mathcal{R}_{\mathcal{A}}(\mathcal{P})$.* □

Example 3.113. Consider program $\mathcal{P}_{3.105}$ of Example 3.105, whose extended argument graph is shown in Figure 3.11 and let $\mathcal{A} = \{\mathtt{p}[2]\}$. Below we show that $\mathtt{p}[1]$ is $\mathcal{A}$-size-restricted in $\mathcal{P}_{3.105}$. Since $\mathtt{p} \in pred(\mathcal{C}_1)$ and $\mathtt{p} \in pred(\mathcal{C}_2)$, we first need to check if $\mathtt{p}[1]$ is $\mathcal{A}$-size-restricted in $\mathcal{C}_1$ and $\mathcal{C}_2$. Since $\mathcal{C}_1 = \{r_1\}$ and r_1 is not $\mathcal{A}$-relevant, we can easily conclude that $\mathtt{p}[1]$ is $\mathcal{A}$-size-restricted in $\mathcal{C}_1$. In the case of $\mathcal{C}_2 = \{r_2\}$, where r_2 is $\mathcal{A}$-relevant, we consider the (only) linear constraint associated with r_2, which is $\overline{\alpha}_p \cdot (1+x, y) \geq \overline{\alpha}_p \cdot (x, 1+y)$. Given $\overline{\alpha}_p = (1,1)$, the constraint is satisfied for all $x, y \in \mathbb{N}$, and since $\overline{\alpha}_p[1] \neq 0$, then $\mathtt{p}[1]$ is $\mathcal{A}$-size-restricted also in $\mathcal{C}_2$.

We now just need to check if for every SCC $\mathcal{C}_l$ such that $\mathtt{p} \in pred(\mathcal{C}_l)$, $\mathtt{p}[1/l]$ only depends on extended arguments $\mathtt{q}[j/k]$ such that $\mathtt{q}[j]$ is $\mathcal{A}$-size-restricted in $\mathcal{C}_k$. Considering $\mathcal{C}_1$, we have that $\mathtt{p}[1/1]$ depends only on itself (see Figure 3.11). Concerning $\mathcal{C}_2$, we have that $\mathtt{p}[1/2]$ depends on itself and

`p[1/1]`. Since `p[1]` is $\mathcal{A}$-size-restricted in both $\mathcal{C}_1$ and $\mathcal{C}_2$, we can conclude that `p[1]` is $\mathcal{A}$-size-restricted in $\mathcal{P}_{3.105}$.

Likewise, it can be easily verified that all other arguments of $\mathcal{P}_{3.105}$ are $\mathcal{A}$-size-restricted in $\mathcal{P}_{3.105}$ as well. □

Correctness

The main goal of this section is to prove the following desired result of soundness.

Theorem 3.114. *Let $\mathcal{P}$ be a program and $\mathcal{A}$ be a set of limited arguments of $\mathcal{P}$. Every $\mathcal{A}$-size-restricted argument of $\mathcal{P}$ is limited. If $\mathcal{P}$ is $\mathcal{A}$-size-restricted then it is limited.* □

We start by introducing some notation. Recall that for normal positive programs $\mathcal{P}$, the notion of limitedness of $\mathcal{P}$ (i.e. when $\mathcal{P}$ is terminating) can be equivalently defined by considering the immediate consequence operator $T_{\mathcal{P}}$. In particular, $\mathcal{P}$ is terminating if for every finite set of facts D, there is a finite natural number n such that $T^n_{\mathcal{P}\cup D}(\varnothing) = T^{\infty}_{\mathcal{P}\cup D}(\varnothing)$.

We will denote $T^i_{\mathcal{P}}(\varnothing)$ simply as $T^i_{\mathcal{P}}$, for every $i \geq 1$. A *labeling* for a predicate symbol p of arity n is a string $\lambda = \lambda_1 ... \lambda_n$, where each $\lambda_i \in \{0, 1\}$. The *projection* of an atom $A = p(t_1, ..., t_n)$ on λ is the atom obtained from A by replacing p with p^{λ} and by deleting every term t_i such that $\lambda_i = 0$, for $1 \leq i \leq n$. For instance, the projection of $p(X, f(X), a, Y)$ on 0101 is $p^{0101}(f(X), Y)$.

A *labeling* for a program $\mathcal{P}$ is a function Λ that associates every p in $pred(\mathcal{P})$ with a labeling for p. The *projection* of an atom A of the form $p(t_1, ..., t_n)$ on Λ, denoted A^{Λ}, simply is the projection of A on $\Lambda(p)$. The *projection* of a program $\mathcal{P}$ (resp. set of atoms I) on Λ, denoted $\mathcal{P}^{\Lambda}$ (resp. I^{Λ}), is the program (resp. set) obtained from $\mathcal{P}$ (resp. I) by replacing every atom A with A^{Λ}.

Consider an argument $p[i] \in args(\mathcal{P})$ with $\Lambda(p) = \lambda = \lambda_1 ... \lambda_n$. Notice that if λ_i $(1 \leq i \leq n)$ is the j-th 1 in λ, then $p^{\lambda}[j]$ is an argument in $args(\mathcal{P}^{\Lambda})$; we say that $p[i]$ is the argument of $\mathcal{P}$ *corresponding to* argument $p^{\lambda}[j]$ of $\mathcal{P}^{\Lambda}$, and vice versa, denoted as $p[i] \rightleftharpoons p^{\lambda}[j]$. As an example, if p is a predicate symbol of $\mathcal{P}$ of arity 4 and $\Lambda(p) = 0101$, then $p[2] \rightleftharpoons p^{0101}[1]$ and $p[4] \rightleftharpoons p^{0101}[2]$.

Proposition 3.115. *Let $\mathcal{P}$ be a program and Λ a labeling for $\mathcal{P}$. For every finite set of facts D, if $M = \mathcal{MM}(\mathcal{P} \cup D)$ and $N = \mathcal{MM}((\mathcal{P} \cup D)^{\Lambda})$, then $M^{\Lambda} \subseteq N$.*

Proof. We show that $(T^i_{\mathcal{P}\cup D})^\Lambda \subseteq T^i_{(\mathcal{P}\cup D)^\Lambda}$ for every $i \geq 1$. The proof is by induction on i.
Base case ($i = 1$). It straightforwardly follows from the observation that both $T^1_{\mathcal{P}\cup D}$ and $T^1_{(\mathcal{P}\cup D)^\Lambda}$ contain only the facts' head.
Inductive step ($i \to i+1$). Consider an atom $A \in T^{i+1}_{\mathcal{P}\cup D}$. By definition of $T_{\mathcal{P}\cup D}$, there is a (ground) rule $A \leftarrow B_1, ..., B_n$ in $ground(\mathcal{P} \cup D)$ s.t. $\{B_1, ..., B_n\} \subseteq T^i_{\mathcal{P}\cup D}$. By the induction hypothesis, $\{B_1^\Lambda, ..., B_n^\Lambda\} \subseteq T^i_{(\mathcal{P}\cup D)^\Lambda}$. Also, notice that $A^\Lambda \leftarrow B_1^\Lambda, ..., B_n^\Lambda$ is a (ground) rule in $ground((\mathcal{P} \cup D)^\Lambda)$. Hence, $A^\Lambda \in T^{i+1}_{(\mathcal{P}\cup D)^\Lambda}$. □

Corollary 3.116. *Let $\mathcal{P}$ be a program and Λ a labeling for $\mathcal{P}$. If $\mathcal{P}^\Lambda$ is terminating, then for every argument $p[i] \in args(\mathcal{P})$, if there exists $p^\lambda[j] \in args(\mathcal{P}^\Lambda)$ such that $p[i] \rightleftharpoons p^\lambda[j]$, then $p[i]$ is limited.*

Proof. Suppose $\mathcal{P}^\Lambda$ is terminating. Let D be a finite set of facts, $M = \mathcal{MM}(\mathcal{P} \cup D)$, and $N = \mathcal{MM}((\mathcal{P} \cup D)^\Lambda)$. Since $\mathcal{P}^\Lambda$ is terminating, then N is finite. By Proposition 3.115, we have that $M^\Lambda \subseteq N$, whence the claim follows. □

We now extend the notion of program expansion introduced in Section 3.6.1. Given a program $\mathcal{P}$, we define $\omega(\mathcal{P}) = \{\,\{\overline{\omega}_{p_1}, ..., \overline{\omega}_{p_n}\} \mid pred(\mathcal{P}) = \{p_1, ..., p_n\}$ and $\overline{\omega}_{p_i} \in \mathbb{Z}^{arity(p_i)}$ for every $1 \leq i \leq n\}$.

Definition 3.117 (Program expansion). *Let $\mathcal{P}$ be a program and $\omega \in \omega(\mathcal{P})$. Given an atom $A = p(t_1, ..., t_m)$ occurring in $\mathcal{P}$, we define $ex(A, \omega) = p(\bar{t}_1, ..., \bar{t}_m)$, where each $\bar{t}_j$ is the sequence $t_j, ..., t_j$ of length $|\overline{\omega}_p[j]| + 1$. Moreover, $ex(\mathcal{P}, \omega)$ denotes the program derived from $\mathcal{P}$ by replacing every atom A with $ex(A, \omega)$.* □

Lemma 3.118. *For every program $\mathcal{P}$ and every $\omega \in \omega(\mathcal{P})$, $\mathcal{P}$ is terminating iff $ex(\mathcal{P}, \omega)$ is terminating.*

Proof. Straightforward from Definition 3.117 and Lemma 3.118. □

Consider a program $\mathcal{P}$ and a set $\mathcal{A}$ of limited arguments of $\mathcal{P}$. Let $\mathcal{C}$ be an SCC of $\mathcal{P}$. We first show that if an argument of $\mathcal{C}$ is $\mathcal{A}$-size-restricted in $\mathcal{C}$ then it is a limited argument of $\mathcal{C}$ (i.e., when $\mathcal{C}$ is considered in isolation as a program). Then, we will show correctness of the technique for an entire program.

Given a program $\mathcal{P}$ and set of limited arguments $\mathcal{A}$, we use $Rel(\mathcal{P}, \mathcal{A})$ to denote the set of all $\mathcal{A}$-relevant rules in $\mathcal{P}$.

Lemma 3.119. *Let $\mathcal{P}$ be a program and $\mathcal{A}$ be a set of limited arguments of $\mathcal{P}$. Let $\mathcal{C}$ be an SCC of $\Sigma(\mathcal{P})$. An argument is limited in $\mathcal{C}$ iff it is limited in $Rel(\mathcal{C}, \mathcal{A})$.*

Proof. The claim follows from the fact that we can derive only a finite number of ground atoms using the rules in $ground(\mathcal{C}) \setminus ground(Rel(\mathcal{C}, \mathcal{A}))$ starting from a finite set of facts—recall that, by definition, all the variables in the head of a non-relevant rule can take only a finite number of values because they appear in the body in correspondence of limited arguments or in atoms that are not mutually recursive with the head. □

Consider a program $\mathcal{P}$ and a set $\mathcal{A}$ of limited arguments of $\mathcal{P}$. Let $\mathcal{C}$ be an SCC of $\mathcal{P}$. Also, let $p[i]$ be an $\mathcal{A}$-size-restricted argument in $\mathcal{C}$ satisfying Condition 2 of Definition 3.107, and let α be a set of vectors satisfying such a condition. Recall that α contains one vector $\overline{\alpha}_q$ for each $q \in pred(\mathcal{C})$.

We define the labeling Λ_α for $\mathcal{C}$ as follows:

- $\Lambda(p) = \lambda_1 ... \lambda_n$ where $\lambda_i = 1$, and for every $j \neq i$, $\lambda_j = 0$ if $\overline{\alpha}_p[j] = 0$, otherwise $\lambda_j = 1$ $(1 \leq j \leq n)$;
- for every $q \in (pred(\mathcal{C}) - \{p\})$, then $\Lambda(q) = \lambda_1 ... \lambda_m$ where $\lambda_j = 0$ if $\overline{\alpha}_q[j] = 0$, otherwise $\lambda_j = 1$ $(1 \leq j \leq m)$.

We also define $\mathcal{A}_\alpha = \{p^\lambda[j] \mid p^\lambda[j] \in args(\mathcal{C}^{\Lambda_\alpha})$ and there exists $p[i] \in \mathcal{A}$ s.t. $p[i] \rightleftharpoons p^\lambda[j]\}$.

Lemma 3.120. *Consider a program $\mathcal{P}$ and a set $\mathcal{A}$ of limited arguments of $\mathcal{P}$. Consider an SCC $\mathcal{C}$ of $\mathcal{P}$. Suppose $p[i]$ is an $\mathcal{A}$-size-restricted argument in $\mathcal{C}$ because of Condition 2 of Definition 3.107, and α is a set of vectors satisfying the condition of Definition 3.107. The argument $p^\lambda[j]$ of $\mathcal{C}^{\Lambda_\alpha}$ s.t. $p[i] \rightleftharpoons p^\lambda[j]$ is an $\mathcal{A}_\alpha$-size-restricted argument in $\mathcal{C}^{\Lambda_\alpha}$.*

Proof. Consider the argument $p^\lambda[j]$ of $\mathcal{C}^{\Lambda_\alpha}$ such that $p[i] \rightleftharpoons p^\lambda[j]$. Recall that α contains one vector $\overline{\alpha}_q$ for each $q \in pred(\mathcal{C})$. Notice that for every $q[k] \in args(\mathcal{C})$, if $q[k] \neq p[i]$ and there does not exist an argument $q^\lambda[l] \in args(\mathcal{C}^{\Lambda_\alpha})$ s.t. $q[k] \rightleftharpoons q^\lambda[l]$, then $\overline{\alpha}_q[k] = 0$. It can be easily verified that if we get rid of all such $\overline{\alpha}_q[k]$'s (which equal 0) from the constraints of Definition 3.107, then we get the set of constraints needed to check if $p^\lambda[j]$ is an $\mathcal{A}_\alpha$-size-restricted argument of $\mathcal{C}^{\Lambda_\alpha}$ and such constraints are obviously satisfied (because $p[i]$ is an $\mathcal{A}$-size-restricted argument of $\mathcal{C}$). □

Let $\mathcal{P}$ be a program and $\mathcal{A}$ be a set of limited arguments of $\mathcal{P}$. The *unary domain* of an argument $p[i] \in args(\mathcal{P})$ w.r.t. $\mathcal{A}$, denoted $\mathbb{U}_{\mathcal{A}}(p[i])$, is $\{-1, 1\}$ if $p[i] \in \mathcal{A}$, and $\{1\}$ otherwise. The *unary domain* of a predicate symbol p of arity n is $\mathbb{U}_{\mathcal{A}}(p) = \mathbb{U}_{\mathcal{A}}(p[1]) \times \cdots \times \mathbb{U}_{\mathcal{A}}(p[n])$. □

An $\mathcal{A}$-*unary-size-restricted* argument $p[i]$ in an SCC is defined in the same way as an $\mathcal{A}$-size-restricted argument in an SCC (cf. Definition 3.107) except that $\mathbb{U}_{\mathcal{A}}(q)$ is used in place of $\mathbb{D}_{\mathcal{A}}(q)$ and $\mathbb{U}_{\mathcal{A}}(p[i]) = \{-1, 0, 1\}$.

The *global size* of an atom $A = p(t_1, ..., t_n)$, denoted $gs(A)$, is $\sum_{i=1}^{n} size(t_i)$.

Lemma 3.121. *Let $\mathcal{P}$ be a program and $\mathcal{A}$ be a set of limited arguments of $\mathcal{P}$. Consider an SCC $\mathcal{C}$ of $\mathcal{P}$. If an argument $p[i]$ of $\mathcal{C}$ is $\mathcal{A}$-unary-size-restricted in $\mathcal{C}$, then $p[i]$ is limited in $\mathcal{C}$.*

Proof. If $p[i]$ is $\mathcal{A}$-unary-size-restricted in $\mathcal{C}$ because Condition 1 of Definition 3.107 is satisfied, then $p[i]$ is limited in $\mathcal{C}$ simply because for every rule in $\mathcal{C}$ having p in the head the values that can be propagated in correspondence of $p[i]$ are taken from limited arguments and thus there is only a finite number of values $p[i]$ can take.

Let us consider now the (more involved) case that $p[i]$ is $\mathcal{A}$-unary-size-restricted in $\mathcal{C}$ because of Condition 2 of Definition 3.107, and let α be a set of vectors satisfying the condition. Recall that α contains one vector $\overline{\alpha}_q$ for each $q \in pred(\mathcal{C})$. Recall also that for each rule, one body atom has been selected to satisfy the inequalities—we call such an atom *the selected body atom*. Below we show that $\mathcal{C}$ is terminating (which in turn implies that $p[i]$ is limited in $\mathcal{C}$).

Given an atom A and a ground instance A' of A, let θ be a unifier of A and A' of the form $\{X_1/t_1, ..., X_m/t_m\}$ where the X_j's are exactly the logical variables occurring in A and all the t_j's are ground terms. It is easy to see that $size(A')$ can be obtained from $size(A)$ by replacing every integer variable x_j in $size(A)$ with $size(t_j)$—obviously, in the same way we can get $gs(A')$ from $gs(A)$. As a consequence, since $p[i]$ is $\mathcal{A}$-unary-size-restricted, then for every ground rule $r \in ground(\mathcal{C})$, there is an atom B in $body(r)$ s.t. if $pr(head(r)) = q$ and $pr(body(r)) = s$, then $\overline{\alpha}_s \cdot size(B) \geq \overline{\alpha}_q \cdot size(head(r))$; furthermore, if $q = p$ and $\overline{\alpha}_p[i] = 0$ then the (stricter) constraint $\overline{\alpha}_s \cdot size(body(r)) > \overline{\alpha}_p \cdot size(head(r))$ holds.

Suppose by contradiction that $\mathcal{C}$ is not terminating. Then, there must exist a finite set of facts D and an infinite sequence of ground atoms $\phi = A_1, A_2, ...$ containing infinitely many distinct atoms s.t. $A_1 \in D$ and for every $j \geq 1$, there is a ground rule $A_{j+1} \leftarrow body$ in $ground(\mathcal{C} \cup D)$ s.t. $body \subseteq \mathcal{MM}(\mathcal{C} \cup D)$,

$A_j \in body$, and A_j is the selected body atom. This means that there is an infinite subsequence $\varphi = A'_1, A'_2, ...$ of ϕ s.t. all atoms have the same predicate symbol q and there exists an m that satisfies the following condition: for every atom A'_j with $j > m$ there is an atom A'_k s.t. $k \leq m$ and A'_k has the same terms of A'_j on the arguments of q in $\mathcal{A}$. Let $A'_j = q(t_1, ..., t_n)$ and $A'_k = q(u_1, ..., u_n)$.

Two cases may occur: either *(a)* $\overline{\alpha}_p[i] \neq 0$ or *(b)* $\overline{\alpha}_p[i] = 0$.

(a) We define $gs_{max} = \max\{gs(A'_l) \mid l \leq m\}$. Then,

$$\overline{\alpha}_q \cdot size(A'_k) \geq \overline{\alpha}_q \cdot size(A'_j) \tag{3.3}$$

$$\sum_{l=1}^{n} \overline{\alpha}_q[l] \cdot size(u_l) \geq \sum_{l=1}^{n} \overline{\alpha}_q[l] \cdot size(t_l) \tag{3.4}$$

$$\sum_{1 \leq l \leq n \wedge q[l] \notin \mathcal{A}} \overline{\alpha}_q[l] \cdot size(u_l) \geq \sum_{1 \leq l \leq n \wedge q[l] \notin \mathcal{A}} \overline{\alpha}_q[l] \cdot size(t_l) \tag{3.5}$$

$$\sum_{1 \leq l \leq n \wedge q[l] \notin \mathcal{A}} size(u_l) \geq \sum_{1 \leq l \leq n \wedge q[l] \notin \mathcal{A}} size(t_l) \tag{3.6}$$

Inequality (3.3) holds as a consequence of the observations made at the beginning of the proof. Then, inequality (3.3) can be rewritten as (3.4). Then, inequality (3.5) follows from the fact that $t_l = u_l$ if $q[l] \in \mathcal{A}$—recall that A'_k and A'_j have the same terms on the arguments of q in $\mathcal{A}$. Finally, inequality (3.6) holds because $\overline{\alpha}_q[l] = 1$ if $q[l] \notin \mathcal{A}$. As $t_l = u_l$ for every $q[l] \in \mathcal{A}$, then $gs(A'_k) \geq gs(A'_j)$. Hence, $gs_{max} \geq gs(A'_j)$.

Thus, for every atom A'_j with $j > m$, it is the case that $gs_{max} \geq gs(A'_j)$. Since programs are range-restricted, then all ground atoms in $\mathcal{MM}(\mathcal{C} \cup D)$ are built from constants and function symbols appearing in $\mathcal{C} \cup D$, which are finitely many. These observations and the definition of gs imply that we can have only finitely many ground q-atoms in $\mathcal{MM}(\mathcal{C} \cup D)$, which is a contradiction.

(b) The same reasoning of case *(a)* applies if there is no p-atom between A'_j and A'_k in ϕ. Otherwise, for the same reasons of case *(a)*, the following inequality holds:

$$\sum_{1 \leq l \leq n \wedge q[l] \notin \mathcal{A}} size(u_l) > \sum_{1 \leq l \leq n \wedge q[l] \notin \mathcal{A}} size(t_l)$$

If $q \neq p$, then the same reasoning of case *(a)* can be applied. Suppose $q = p$. Let $gs_{max} = \max\{\sum_{1 \leq h \leq n \wedge h \neq i \wedge q[h] \notin \mathcal{A}} size(v_h) \mid A'_l = p(v_1, ..., v_n) \wedge l \leq m\}$.

Observe that φ contains an infinite subsequence σ of distinct atoms with the same values on the limited arguments.

For every pair of atoms $A_{\ell_1} = p(v_1, ..., v_n)$ and $A_{\ell_2} = p(v'_1, ..., v'_n)$ in σ s.t. $\ell_2 > \ell_1 > m$, the following holds:

$$\sum_{1 \le l \le n \wedge l \ne i \wedge q[l] \notin \mathcal{A}} size(v'_l) > \sum_{1 \le l \le n \wedge l \ne i \wedge q[l] \notin \mathcal{A}} size(v_l) > gs_{max}$$

which means that, for every atom $p(s_1, ..., s_n)$ in σ the summation $\sum_{1 \le l \le n \wedge l \ne i \wedge q[l] \notin \mathcal{A}} size(s_l)$ strictly decreases w.r.t. the previous atoms in the sequence σ and thus the number of atoms in σ is bounded by gs_{max}, and thus finite, which is a contradiction. Hence, $\mathcal{C}$ is terminating, which in turn implies that $p[i]$ is limited in $\mathcal{C}$. □

Lemma 3.122. *Let $\mathcal{P}$ be a program and $\mathcal{A}$ be a set of limited arguments of $\mathcal{P}$. Consider an SCC $\mathcal{C}$ of $\mathcal{P}$. If an argument $p[i]$ of $\mathcal{C}$ is $\mathcal{A}$-size-restricted in $\mathcal{C}$, then $p[i]$ is limited in $\mathcal{C}$.*

Proof. If $p[i]$ is $\mathcal{A}$-size-restricted in $\mathcal{C}$ because Condition 1 of Definition 3.107 is satisfied, then $p[i]$ is limited in $\mathcal{C}$ simply because for every rule in $\mathcal{C}$ having p in the head the values that can be propagated in correspondence of $p[i]$ are taken from limited arguments and thus there is only a finite number of values $p[i]$ can take.

Let us consider now the case that $p[i]$ is $\mathcal{A}$-size-restricted in $\mathcal{C}$ because of Condition 2 of Definition 3.107, and let α be a set of vectors satisfying the condition. Then, $Rel(\mathcal{C}, \mathcal{A})$ satisfies the condition as well. Let $p^\lambda[j]$ be the argument of $Rel(\mathcal{C}, \mathcal{A})^{\Lambda_\alpha}$ s.t. $p[i] \rightleftharpoons p^\lambda[j]$. By Lemma 3.120, $p^\lambda[j]$ is an $\mathcal{A}_\alpha$-size-restricted argument of $Rel(\mathcal{C}, \mathcal{A})^{\Lambda_\alpha}$. Then, $p^\lambda[j]$ is unary-size-restricted in $ex(Rel(\mathcal{C}, \mathcal{A})^{\Lambda_\alpha}, \alpha)$. As shown in the proof of Lemma 3.121, $ex(Rel(\mathcal{C}, \mathcal{A})^{\Lambda_\alpha}, \alpha)$ is terminating. By Lemma 3.118, $Rel(\mathcal{C}, \mathcal{A})^{\Lambda_\alpha}$ is terminating. By Corollary 3.116, $p[i]$ is limited in $Rel(\mathcal{C}, \mathcal{A})$. By Lemma 3.119, $p[i]$ is limited in $\mathcal{C}$. □

Complexity and Expressivity

In this section, we provide results on the complexity and the expressivity of the class of $\mathcal{A}$-size-restricted programs.

We start by showing that checking if an argument is $\mathcal{A}$-size-restricted in an SCC is in *NP*.

Theorem 3.123. *Let $\mathcal{P}$ be a program and $\mathcal{A}$ be a set of limited arguments of $\mathcal{P}$. Given an SCC $\mathcal{C}$ of $\mathcal{P}$, checking whether an argument of $\mathcal{C}$ is $\mathcal{A}$-size-restricted in $\mathcal{C}$ is in NP.*

Proof. We will first show that the *NP* bound holds for programs whose rules have at most one atom in the body. Then, the result is naturally extended to general programs.

Let $p[i]$ be an argument of $\mathcal{C}$ for which we want to check if it is $\mathcal{A}$-size-restricted and let $R = R_1 \cup R_2 = \{r_1, ..., r_m\}$ be the set of $\mathcal{A}$-relevant rules of $\mathcal{C}$ where $r \in R_1$ iff $pred(head(r)) = p$. By following Definition 3.107, we can check if $p[i]$ is $\mathcal{A}$-size-restricted in $\mathcal{C}$ by either checking Condition 1 or checking Condition 2. Condition 1 requires to verify for every rule $r \in \mathcal{C}$ (which are linear in number w.r.t. $||\mathcal{P}||$) with $head(r) = p(t_1, ..., t_k)$, if t_j has a variable in some body argument belonging to $\mathcal{A}$. This check can be trivially carried on by a linear scanning of the rule's symbols and the elements of $\mathcal{A}$. Considering Condition 2, for each $1 \leq k \leq m$, let $B_k = body(r_k)$, $H_k = head(r_k)$, and let $lc_{r_k}(\rhd)$ be the linear constraint associated to r_k of the form

$$\overline{\alpha}_{pred(B_k)} \cdot size(B_k) \rhd \overline{\alpha}_{pred(H_k)} \cdot size(H_k)$$

where $\rhd \in \{>, \geq\}$. In order to check whether Condition 2 is satisfied, we need to find an assigment, for each $q \in pred(\mathcal{C})$, to vector $\overline{\alpha}_q$, such that $\overline{\alpha}_q \in \mathbb{D}_{\mathcal{A}}(q)$ and such that $lc_{r_k}(\geq)$ is satisfied for every $1 \leq k \leq m$. Then, we verify whether either $\overline{\alpha}_p[i] \neq 0$ or $lc_{r_k}(>)$ is satisfied whenever $pred(head(r_k)) = p$. We now introduce some results that will allow us to rewrite the above problem into an equivalent one, which is more suitable for our discussion.

First, note that every constraint $lc_{r_k}(\rhd)$ can be rewritten in the form $\gamma_{k_1} \cdot x_1 + ... + \gamma_{k_l} \cdot x_l + \gamma_{k_0} \rhd 0$, where the x_j's are the integer variables occurring in $lc_{r_k}(\rhd)$ and each γ_{k_j} is an integer only depending on vectors $\overline{\alpha}_{pred(B_k)}$ and $\overline{\alpha}_{pred(H_k)}$.

Moreover, an important property of such constraints holds.

Lemma 3.124. *Let $\gamma_0, ..., \gamma_n \in \mathbb{Z}$ and let $\rhd \in \{>, \geq\}$. The inequality $\gamma_1 \cdot x_1 + ... + \gamma_n \cdot x_n + \gamma_0 \rhd 0$ is satisfied for every $x_1, ..., x_n \in \mathbb{N}$ iff for $1 \leq j \leq n$, $\gamma_j \geq 0$ and $\gamma_0 \rhd 0$.*

Proof.
($\Leftarrow$) Straightforward.
($\Rightarrow$) Note that $\gamma_0 \rhd 0$ must be true, since in the case where $x_j = 0$, $1 \leq j \leq n$,

the inequality coincides with $\gamma_0 \rhd 0$. Then, assume that $\gamma_j < 0$, for some $1 \leq j \leq n$, then the inequality is not satisfied for $x_h = 0$ for all $h \neq j$ and $x_j = \lfloor abs(\frac{\gamma_0}{\gamma_j}) \rfloor + 1$. □

The lemma above allows us to rewrite every constraint of the form $lc_{r_k}(\rhd) = \gamma_{k_1} \cdot x_1 + ... + \gamma_{k_l} \cdot x_l + \gamma_{k_0} \rhd 0$ into an equivalent set of constraints

$$con_{r_k}(\rhd) = \{\gamma_{k_1} \geq 0, ..., \gamma_{k_l} \geq 0, \gamma_{k_0} \rhd 0\}$$

Furthermore, for each $q \in pred(\mathcal{C})$, $\mathbb{D}_{\mathcal{A}}(q[j])$ ($1 \leq j \leq arityq$) is either the set $\mathbb{N}$ or the set $\mathbb{Z}$. Thus, every expression $\overline{\alpha}_q \in \mathbb{D}_{\mathcal{A}}(q)$ can be rewritten into an equivalent set of $arityq$ linear constraints where, for $1 \leq j \leq arityq$, the constraint is $\overline{\alpha}_q[j] \geq 0$ if $\mathbb{D}_{\mathcal{A}}(q[j]) = \mathbb{N}$, otherwise the constraint is $\overline{\alpha}_q[j] \gtreqless 0$ (i.e., the j-th component is free). Let Dom be the set of all linear constraints obtained by rewriting every expression $\overline{\alpha}_q \in \mathbb{D}_{\mathcal{A}}(q)$, for each $q \in pred(\mathcal{C})$.

With the result above and Lemma 3.124 the problem of checking whether $p[i]$ is $\mathcal{A}$-size-restricted in $\mathcal{C}$ is equivalent to finding a solution for one of the following three sets of linear constraints.

1. $Con_1 = \bigcup_{k=1}^{m} con_{r_k}(\geq) \cup Dom \cup \{\overline{\alpha}_p[i] > 0\}$;
2. $Con_2 = \bigcup_{k=1}^{m} con_{r_k}(\geq) \cup Dom \cup \{\overline{\alpha}_p[i] < 0\}$;
3. $Con_3 = \bigcup_{r \in R_1} con_r(>) \cup \bigcup_{r \in R_2} con_r(\geq) \cup Dom$;

where the only unknowns are the components of vectors $\overline{\alpha}_q$, for each $q \in pred(\mathcal{C})$.

It is easy to see that $p[i]$ is $\mathcal{A}$-size-restricted in $\mathcal{C}$ iff Con_1, Con_2 or Con_3 admit a solution. In particular, the first set Con_1 represents the case when every linear constraint $lc_{r_k}(\geq)$ is satisfied and $\overline{\alpha}_p[i] \neq 0$, more specifically, $\overline{\alpha}_p[i] > 0$. The second set Con_2, is identical, but it takes into account the case where $\overline{\alpha}_{p_i}[j] < 0$. These two constraint essentially check for Condition 2(a) and the first part of Condition 2(b) of Definition 3.107. The last set Con_3, instead, takes into account the case where $\overline{\alpha}_p[i]$ could be zero, but for every rule r in R_1 (i.e., the rules defining p), the constraints $lc_r(>)$ Must be satisfied. In a nutshell, Con_3 checks for Condition 2(a) and the second part of Condition 2(b) of Definition 3.107.

What remains to show is the complexity of constructing the above sets and the complexity of checking whether one of such sets admit a solution. Note that if v is the maximum number of distinct variables occurring in a rule in

R and a is the number of arguments in $pred(\mathcal{C})$, for every rule $r \in R$, the set $con_r(\triangleright)$ has at most $v+1$ constraints and the set Dom has a constraints. Thus, each Con_j has at most $m \cdot (v+1) + a + 1$ linear constraints, with at most $2 \cdot a$ unknowns per constraint. Clearly, the size of each Con_j is bounded by $O(||\mathcal{P}||)$. Regarding the complexity of finding a solution to one of the sets Con_j, note that the constraints in each Con_j are always of one of the following four forms: $>, <, \geq, \gtreqless$. It is well known that constraints of the form $<$ and $\gtreqless$ can be rewritten to constraints of the form $>$ and of the form $\geq$ respectively by (linearly) increasing the number of unknowns. Finally, checking whether a set of linear constraints of the form $>, \geq$ admits a solution, where the unknowns are restricted to be integers, is in *NP* [41].

In the case of general programs, we can exploit the following non deterministic polynomial time procedure: given the argument $p[i]$, either check in deterministic polynomial time if it satisfies Condition 1 of Definition 3.107 or guess a body atom B_r for each rule $r \in R$ and construct the program $\mathcal{C}' = \bigcup_{r \in R} \{head(r) \leftarrow B_r\}$. Then, check whether Condition 2 is satisfied in non-deterministic polynomial time, as shown for programs whose rules have only one atom in the body. □

From the theorem above, we obtain that checking whether a program is $\mathcal{A}$-size-restricted is in *NP*.

Theorem 3.125. *Let $\mathcal{P}$ be a program and $\mathcal{A}$ be a set of limited arguments of $\mathcal{P}$. Checking whether (an argument of) $\mathcal{P}$ is $\mathcal{A}$-size-restricted (in $\mathcal{P}$) is in NP.*

Proof. As in proof of Theorem 4.67, we first assume that our program $\mathcal{P}$ only contains rules with one atom in the body. To show the *NP* upper-bound we present the following procedure:

1. Construct the firing graph of $\mathcal{P}$ and compute the SCCs $\mathcal{C}_1, ..., \mathcal{C}_n$ of $\mathcal{P}$;
2. Construct the extended argument graph of $\mathcal{P}$;
3. Let $Q = \{q[j/k] \mid p[i/l]$ depends on $q[j/k], 1 \leq l \leq n\}$ be the set of extended arguments on which every $p[i/l]$ depends on;
4. Construct for each SCC $\mathcal{C}_j$ $(1 \leq j \leq n)$, the corresponding sets Con_1, Con_2, Con_3 of linear constraints, as shown in the proof of Theorem 4.67. Then rewrite Con_1,Con_2,Con_3 to the three sets $Con_1^j, Con_2^j, Con_3^j$, where their vectors $\overline{\alpha}_b \in \mathbb{D}_{\mathcal{A}}(b)$ are renamed to $\overline{\alpha}_b^j$.
5. Non-deterministically choose, for each $1 \leq j \leq n$, one of the three sets of constraints $Con_{h_j}^j$ $(1 \leq h_j \leq 3)$. For each $1 \leq j \leq n$ and for each $b \in pred(\mathcal{C}_j)$, non-deterministically choose an assignment to vectors $\overline{\alpha}_b^j$;

6. Verify that $p[i]$, for each $1 \le j \le n$ such that $p \in \mathit{pred}(\mathcal{C}_j)$, either satisfies Condition 1 of Definition 3.107 w.r.t. $\mathcal{C}_j$ or that the vectors $\overline{\alpha}_b^j$ satisfy Condition 2 of Definition 3.107 w.r.t. $\mathcal{C}_j$.
7. Finally, check whether for every every $q[j/k] \in Q$, $q[j]$ either satisfies Condition 1 of Definition 3.107 w.r.t. $\mathcal{C}_k$ or that the vectors $\overline{\alpha}_b^k$ satisfy Condition 2 of Definition 3.107 w.r.t. $\mathcal{C}_k$.

The firing graph can be constructed in polynomial time [16] and the SCCs of $\mathcal{P}$ can be obtained by means of the well-known polynomial time Tarjan's algorithm. The extended argument graph can be constructed in polynomial time as well, since the maximum number of edges $(q[j/k], b[h/l])$ is bounded by $(a \cdot n)^2$, where a is the maximum arity of the predicate symbols of $\mathcal{P}$. Moreover, according to the definition of extended argument graph, checking whether an edge $(q[j/k], b[h/l])$ belongs to the extended argument graph is trivially feasible in polynomial time (recall that checking whether two atoms unify is in polynomial time too [78]). As shown in the proof of Theorem 4.67, also the construction of the sets Con_1, Con_2, Con_2 can be accomplished in polynomial time. Step 5 requires non-deterministic polynomial time, since the satisfiability problem of the constraints $Con^j_{h_j}$ is in NP, as shown in the proof of Theorem 4.67. The last two steps simply verify the conditions of Definition 3.107, which has already been shown to be feasible in polynomial time as well. □

Recall that AR, BP, RB, and SR denote, respectively, the set of all argument-restricted [60], bounded [52], rule-bounded [16], and $\varnothing$-size-restricted programs. Moreover, given two sets A and B, we use $A \nparallel B$ as a shorthand for $A \not\subseteq B \wedge B \not\subseteq A$. The following theorem compares the new approach with well-known terminating classes previously proposed.

Theorem 3.126. *$AR \nparallel SR$, $RB \subsetneq SR$, and $BP \nparallel SR$*

Proof. To show that $AR \nparallel SR$ (resp. $BP \nparallel SR$) it suffices to show that there exists a program which is in AR (resp. BP) but not in SR and vice versa. The program $\mathcal{P} = \{\mathtt{q(X)} \leftarrow \mathtt{p(f(X))}.\ \mathtt{p(f(X))} \leftarrow \mathtt{q(X)}.\}$ is in AR (resp. BP) but it is not in SR. Conversely, the program $\mathcal{P}_{4.51}$ of Example 4.51 is in SR but it is not in AR (resp. BP).

We now show that $RB \subseteq SR$. We say that an SCC $\mathcal{C}$ of a program $\mathcal{P}$ is *non-trivial* if there exists at least one edge in $\Sigma(\mathcal{P})$ between two (not necessarily distinct) nodes of $\mathcal{C}$. An SCC is *trivial* if it is not non-trivial. More-

over, a rule $r \in \mathcal{P}$ is *relevant* if the set of atoms $body(r) \setminus rbody(r)$ does not contain all logical variables in $head(r)$.[3] Let $\mathcal{P}$ be a rule-bounded program, then every non-trivial SCC $\mathcal{P}$ is rule-bounded [16], where a non-trivial SCC $\mathcal{C}$ of $\mathcal{P}$, with $pred(\mathcal{C}) = \{p_1, ..., p_m\}$, is rule-bounded if there exist m vectors $\overline{\alpha}_{p_h} \in (\mathbb{N} \setminus \{0\})^{arity p_h}$ $(1 \leq h \leq m)$ such that, for every relevant rule $r \in \mathcal{C}$, with $H = head(r) = p_i(t_1, ..., t_k)$, there exists an atom $B = p_j(u_1, ..., u_l)$ in $body(r)$ such that

$$\overline{\alpha}_{p_j} \cdot size(B) \geq \overline{\alpha}_{p_i} \cdot size(H) \tag{3.7}$$

holds for every non-negative value of the integer variables in $size(B), size(H)$.

We show that $\mathcal{P}$ is $\varnothing$-size-restricted by induction on the number of SCCs of $\mathcal{P}$.

Base case $(n = 1)$: $\mathcal{P}$ is rule-bounded and has only one SCC $\mathcal{C}$. If $\mathcal{C}$ is trivial, then, it contains only one rule, and this rule is non-relevant, and then it is also not $\varnothing$-relevant, which implies that $\mathcal{P}$ is trivially $\varnothing$-size-restricted. If $\mathcal{C}$ is non trivial, since each vector $\overline{\alpha}_{p_h}$, $1 \leq h \leq n$ has only strictly positive components, and all the constraints in (3.7) are satisfied, every argument of $\mathcal{C}$ is $\varnothing$-size-restricted in $\mathcal{C}$ (Condition 1 of Definition 3.112). Since the arguments of $\mathcal{P}$ are only the ones of $\mathcal{C}$, Condition 2 of Definition 3.112 is satisfied too and thus, $\mathcal{P}$ is $\varnothing$-size-restricted.

Inductive step $(n > 1)$: Let $\mathcal{C}^{(n-1)}$ be a rule-bounded program and let $\mathcal{C}_1, ..., \mathcal{C}_{n-1}$ be its SCCs, where each non-trivial SCC is rule-bounded. Assume that $\mathcal{C}^{(n-1)}$ is $\varnothing$-size-restricted by inductive hypothesis. We now add either a trivial or a non-trivial rule-bounded SCC $\mathcal{C}_n$ to $\mathcal{C}^{(n-1)}$, obtaining the program $\mathcal{C}^{(n)}$. In either case, $\mathcal{C}^{(n)}$ is still rule-bounded. If $\mathcal{C}_n$ is trivial, since $\mathcal{C}^{(n-1)}$ is $\varnothing$-size-restricted by hypothesis, with a similar reasoning of the base step, we conclude that $\mathcal{C}^{(n)}$ is $\varnothing$-size-restricted. If $\mathcal{C}_n$ is non-trivial, since $\mathcal{C}_n$ is rule-bounded, every argument in $\mathcal{C}_n$ is $\varnothing$-size-restricted in $\mathcal{C}_n$, as shown in the base step. Moreover, by inductive hypothesis, every argument in $\mathcal{C}^{(n-1)}$ is $\varnothing$-size-restricted in $\mathcal{C}^{(n-1)}$. With the addition of $\mathcal{C}_n$ to $\mathcal{C}^{(n-1)}$, all the extended arguments $p_h[i/l]$ $(1 \leq l \leq n-1)$ may now depend on the extended arguments $p_k[j/n]$, but they are such that the arguments $p_k[j]$ are $\varnothing$-size-restricted in $\mathcal{C}_n$, by construction of $\mathcal{C}_n$. In turn, every extended argument $p_k[j/n]$ may now depend on arguments $p_h[i/l]$ $(1 \leq l \leq n-1)$, but they are such that the arguments $p_h[i]$ are $\varnothing$-size-restricted in their respective

[3] The notion of relevance coincides with the notion of $\mathcal{A}$-relevance, when $\mathcal{A} = \varnothing$.

SCCs, by inductive hypothesis. We can then conclude that every argument in $\mathcal{C}^{(n)}$ is $\varnothing$-size-restricted in $\mathcal{C}^{(n)}$, and thus $\mathcal{C}^{(n)}$ is $\varnothing$-size-restricted. To show the proper inclusion, please note that the program $\mathcal{P}_{4.51}$ of Example 4.51 is in $Size - Restricted$ but not in RB. □

Note that the new technique strictly generalizes the rule-bounded criterion even when the set of limited arguments $\mathcal{A}$ is empty. Looking at the size of parts of multiple atoms, as opposed to the entire size of a single atom like the rule-bounded criterion does, allows our criterion to include more programs.

By combining our technique with the argument-restricted or bounded criterion we can recognize more limited programs than by using any of them alone. We use $AR + SR$ (resp. $BP + SR$) to denote the set of all $\mathcal{A}$-size-restricted programs where, for each program, $\mathcal{A}$ is the set of its argument-restricted (resp. bounded) arguments.

Corollary 3.127. *$AR \subsetneq AR + SR$, $SR \subsetneq AR + SR$, $BP \subsetneq BP + SR$, $SR \subsetneq BP + SR$.*

Proof. First of all, recall that for every program $\mathcal{P}$, given a set $\mathcal{A}$ of limited arguments of $\mathcal{P}$, $\mathcal{P}$ is $\mathcal{A}$-size-restricted iff $args(\mathcal{P}) = \mathcal{A} \cup \mathcal{R}_{\mathcal{A}}(\mathcal{P})$. Thus, $T \subseteq T{+}SR$, where $T \in \{AR, BP\}$, is straightforward. The property $SR \subseteq T{+}SR$, where $T \in \{AR, BP\}$, follows from the fact that that if $\mathcal{A} \subseteq \mathcal{A}'$ then $\mathcal{R}_{\mathcal{A}}(\mathcal{P}) \subseteq SR\mathcal{A}'\mathcal{P}$. To show that $T \subset T{+}SR$ and $SR \subset T{+}SR$ ($T \in \{AR, BP\}$), consider the program $\mathcal{P} = \mathcal{P}_1 \cup \{\mathtt{s(X)} \leftarrow \mathtt{q(f(X))};\ \mathtt{q(f(X))} \leftarrow \mathtt{s(X)};\ \mathtt{p(X, g(X), g(X))} \leftarrow \mathtt{q(X)}\}$. This program is neither in T nor in SR, but it is in $T{+}SR$ ($T \in \{AR, BP\}$). □

Iterated Criterion

The size-restricted technique presented in the previous section starts from a (possibly empty) set of limited arguments $\mathcal{A}$ and gives as output a new set of limited arguments $\mathcal{A}'$. The question is whether the technique, starting from the resulting set of limited arguments $\mathcal{A}'$, could compute a new set of limited arguments $\mathcal{A}'' \supset \mathcal{A}'$. As shown by the next example, the answer is positive and thus our technique can benefit from an iterative application of itself.

Example 3.128. Consider the following program $\mathcal{P}_{3.128}$.

$$\mathtt{p(f(X), f(Y))} \leftarrow \mathtt{p(X, Y), b(X)}.$$

The program has only one SCC consisting of the rule above. Assume that $\mathcal{A} = \varnothing$. By choosing the first body atom of the rule, we get the following inequality:

$$\overline{\alpha}_p \cdot (x, y) \geq \overline{\alpha}_p \cdot (x{+}1, y{+}1)$$

The vectors $\overline{\alpha}_p = (0, 0)$ and $\overline{\alpha}_b = (1)$ satisfy the conditions of Definition 3.107. Therefore, the resulting set of $\mathcal{A}$-size-restricted arguments is $\mathcal{A}' = \{\mathtt{b}[1]\}$.

Now, considering $\mathcal{A}'$ as the starting set of limited arguments, we determine that $\mathtt{p}[1]$ is limited too, by Condition 1 of Definition 3.107. The new set of limited arguments is $\mathcal{A}'' = \mathcal{A}' \cup \{\mathtt{p}[1]\}$. Finally, considering the vectors $\overline{\alpha}_p = (-1, 1)$ (recall that $\mathtt{p}[1] \in \mathcal{A}''$) and $\overline{\alpha}_b = (0)$, the constraint is satisfied for all non-negative values of its integer variables. Then, $\mathtt{p}[2]$ is limited, $\mathcal{A}''' = \mathcal{A}'' \cup \{\mathtt{p}[2]\}$, and hence $\mathcal{P}_{3.128}$ is limited. □

Thus, we introduce a simple operator that iteratively applies the size-restricted criterion by using at each iteration the limited arguments derived at previous iterations.

Definition 3.129. *Let $\mathcal{P}$ be a program and $\mathcal{A}$ be a set of limited arguments of $\mathcal{P}$. We define the operator $\Psi_{\mathcal{P}}(\mathcal{A}) = \mathcal{A} \cup \mathcal{R}_{\mathcal{A}}(\mathcal{P})$. For $i \geq 1$, we define the i-th iteration of $\Psi_{\mathcal{P}}$ as follows:*

$$\begin{aligned} &\Psi^1_{\mathcal{P}}(\mathcal{A}) = \Psi_{\mathcal{P}}(\mathcal{A}) \\ &\Psi^{i+1}_{\mathcal{P}}(\mathcal{A}) = \Psi_{\mathcal{P}}(\Psi^i_{\mathcal{P}}(\mathcal{A})), \quad \textit{for } i > 1. \end{aligned}$$

□

Obviously, $\Psi^i_{\mathcal{P}}(\mathcal{A}) \subseteq \Psi^{i+1}_{\mathcal{P}}(\mathcal{A})$ for every $i \geq 1$ and since the number of arguments of $\mathcal{P}$ is finite, then there always exists a finite $n \leq |args(\mathcal{P})|$ such that $\Psi^n_{\mathcal{P}}(\mathcal{A}) = \Psi^{n+1}_{\mathcal{P}}(\mathcal{A})$; we denote $\Psi^n_{\mathcal{P}}(\mathcal{A})$ as $\Psi^{\infty}_{\mathcal{P}}(\mathcal{A})$.

Corollary 3.130. *Let $\mathcal{P}$ be a program and $\mathcal{A}$ be a set of limited arguments of $\mathcal{P}$. Every argument in $\Psi^{\infty}_{\mathcal{P}}(\mathcal{A})$ is limited.*

Proof. Straightforward. □

4

Termination of the Chase

In this chapter, we cover the problem of verifying whether whatever some variation of the Chase procedure terminates when a set of TGDs or TGDs and EGDs is given. As already stated, the problem of deciding whether some variation of the Chase terminates is undecidable, in general. Thus, to tackle the problem we can rely on two different approaches. One is to identify subclasses of TGDs and EGDs for which checking termination of the Chase becomes decidable; the other approach is to provide sufficient-only criteria which given a set of TGDs and EGDs Σ, are able to conclude that some variation of the Chase terminates when the set Σ is given.

In the rest of this chapter we formalize the problem of the termination of the Chase, we present the most important sufficient-only criteria known in the literature and in Section 4.3 and Section 4.4 a decidable class of TGDs and a more general sufficient-only condition over TGDs and EGDs are presented, respectively.

4.1 The Chase termination problem

We denote by $\mathsf{CT}^{\star}_{\forall}$, with $\star \in \{\mathsf{std}, \mathsf{obl}, \mathsf{sobl}, \mathsf{core}\}$, the class of sets of TGDs and EGDs Σ such that for every database D all $\star$-chase sequences of D with Σ are terminating. Analogously, we denote by $\mathsf{CT}^{\star}_{\exists}$ the class of sets of dependencies Σ such that for every database D there is a terminating $\star$-chase sequence of D with Σ.

Even focusing on TGDs only, the problem of verifying whether a set of dependencies belongs to $\mathsf{CT}^{\star}_{\forall}$ or $\mathsf{CT}^{\star}_{\exists}$, for $\star \in \{\mathsf{std}, \mathsf{obl}, \mathsf{sobl}, \mathsf{core}\}$, is undecidable [47, 49].

For sets of TGDs only, it has already been shown in [65, 69] that:

$$\mathsf{CT}^{\mathsf{obl}}_{\forall} = \mathsf{CT}^{\mathsf{obl}}_{\exists} \subsetneq \mathsf{CT}^{\mathsf{sobl}}_{\forall} = \mathsf{CT}^{\mathsf{sobl}}_{\exists} \subsetneq \mathsf{CT}^{\mathsf{std}}_{\forall} \subsetneq \mathsf{CT}^{\mathsf{std}}_{\exists} \subsetneq \mathsf{CT}^{\mathsf{core}}_{\forall} = \mathsf{CT}^{\mathsf{core}}_{\exists}$$

The above hierarchy is relevant because if we determine that a set of TGDs belongs to $\mathsf{CT}^{\star}_{\mathsf{q}}$ with $\mathsf{q} \in \{\forall, \exists\}$ and $\star \in \{\mathsf{obl}, \mathsf{sobl}\}$, then Σ belongs to $\mathsf{CT}^{\mathsf{std}}_{\forall}$ (and, of course, $\mathsf{CT}^{\mathsf{std}}_{\exists}$), and, in some cases, the analysis of the oblivious or semi-oblivious Chase is easier. In fact, the importance of these Chase variants has been widely recognized and their behavior has been studied in different works [19, 50, 56, 63, 65]. Thus, hereafter we will focus on the following (semi-)oblivious Chase termination problems.

$\forall$-Sequence $\star$-Chase Termination:
Instance: A set Σ of TGDs and EGDs.
Question: Does $\Sigma \in \mathsf{CT}^{\star}_{\forall}$?

$\exists$-Sequence $\star$-Chase Termination:
Instance: A set Σ of TGDs and EGDs.
Question: Does $\Sigma \in \mathsf{CT}^{\star}_{\exists}$?

As shown above, recall that for TGDs only $\mathsf{CT}^{\mathsf{obl}}_{\forall} = \mathsf{CT}^{\mathsf{obl}}_{\exists} \subset \mathsf{CT}^{\mathsf{sobl}}_{\forall} = \mathsf{CT}^{\mathsf{sobl}}_{\exists}$ [49]. This implies that when only TGDs are considered, the preceding decision problems coincide for the (semi-)oblivious Chase. Thus, when the termination analysis is carried on TGDs only, we will refer to just the $\star$-chase termination problem, and we write $\mathsf{CT}^{\star}$ for the classes $\mathsf{CT}^{\star}_{\forall}$ and $\mathsf{CT}^{\star}_{\exists}$, where $\star \in \{\mathsf{obl}, \mathsf{sobl}\}$.

Another useful notion for our later analysis is the so-called critical database for a set of TGDs [63]. Formally, the *critical database* for a schema $\mathcal{R}$ is the database $D_{\mathsf{c}}(\mathcal{R}) = \{p(\mathsf{c}, \ldots, \mathsf{c}) \mid p \in \mathcal{R} \text{ and } \mathsf{c} \in \mathbf{C}\}$. The critical database for a set Σ of TGDs is defined as the database $D_{\mathsf{c}}(sch(\Sigma))$; for brevity, we will refer to $D_{\mathsf{c}}(sch(\Sigma))$ by $D_{\mathsf{c}}(\Sigma)$. As shown in [63], to check for the termination of the (semi-)oblivious Chase (thus for TGDs only) it suffices to focus on the critical database.

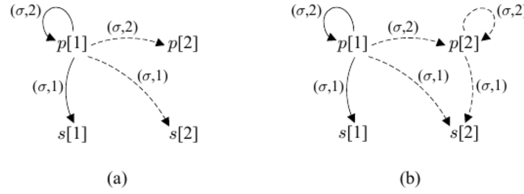


Fig. 4.1: (Extended) dependency graph of Example 4.2.

4.2 State of the art

4.2.1 Weak acyclicity

The class of weakly acyclic TGDs, defined in [38], is one of the first classes of TGDs defined in the literature belonging to $\mathsf{CT}_{\forall}^{\mathsf{std}}$. It is defined via an acyclicity condition on a graph, which encodes how terms are propagated among the positions of the underlying schema during the Chase. In fact, weak-acyclicity uses the well-known dependency graph.

Definition 4.1. *The* dependency graph *of a set Σ of TGDs is a labeled, directed multigraph $DG(\Sigma) = (N, E, \lambda)$, where $N = pos(sch(\Sigma))$, $\lambda : E \rightarrow \Sigma \times \mathbb{N}$, and the edge-set E is as follows: for each $r \in \Sigma$, for each $V \in fr(r)$, and for each $\pi \in pos(body(r), V)$, with $head(r) = A_1, \ldots, A_k$: (1) for each $i \in [k]$, and for each $\pi' \in pos(A_i, V)$, there is a* normal *edge $e = (\pi, \pi') \in E$ with $\lambda(e) = (r, i)$; (2) for each $W \in ex(r,)$, for each $i \in [k]$, and for each $\pi' \in pos(A_i, W)$, there is a* special *edge $e = (\pi, \pi') \in E$ with $\lambda(e) = (r, i)$; (3) no other edges are in E.* □

A set Σ of TGDs is *weakly-acyclic* (resp., *richly-acyclic*) if no cycle in $DG(\Sigma)$ contains a special edge. The class of weakly acyclic TGDs is denoted by WA.

Intuitively, a normal edge (π, π') in the dependency graph keeps track of the fact that a term may propagate from π to π' during the Chase. Moreover, a special edge (π, π'') keeps track of the fact that propagation of a value from π to π' also creates a null value at position π''.

Example 4.2. Consider the set Σ consisting of the TGD

$$r \ = \ p(X, Y) \ \rightarrow \ \exists Z\, s(X, Z), p(X, Z).$$

The graph $DG(\Sigma)$ is depicted in Figure 4.1(a), where the dashed arrows represent special edges. Observe that the normal edges occur due to the variable X, while the special edges due to the existentially quantified variable Z. □

4.2.2 Rich Acyclicity

Rich acyclic TGDs is the class of TGDs defined in [58], for which the oblivious Chase is guaranteed to terminate, that is sets of TGDs in this class belong to $\mathsf{CT}_{\forall}^{\mathsf{obl}} = \mathsf{CT}_{\exists}^{\mathsf{obl}}$. The class is defined over the extended dependency graph.

The extended dependency graph of a set Σ of TGDs, introduced in [58], is obtained from the dependency graph of Σ by adding some additional special edges from the positions where non-frontier variables occur to the positions where existentially quantified variables appear.

A set Σ of TGDs is *richly-acyclic* if no cycle in $EDG(G)$ contains a special edge. The class of richly acyclic TGDs is denoted by RA. It is easy to verify that $\mathsf{RA} \subset \mathsf{WA}$.

The extended dependency graph of Σ given in Example 4.1 is shown in Figure 4.1(b); the additional special edges (dashed arrows) are due to the non-frontier variable Y.

4.2.3 (C-)Stratification

The first extension of weak acyclicity is the stratification criterion, proposed by [33] and aimed to identify classes of TGDs and EGDs belonging to $\mathsf{CT}_{\exists}^{\mathsf{std}}$. The idea behind stratification is to decompose a set of TGDs and EGDs into independent subsets, where each subset consists of constraints that may fire each other, and to check each component separately for weak acyclicity.

Definition 4.3 (Precedence relation). *[33] Given a set of TGDs and EGDs Σ and given two dependencies r_1 and r_2 in Σ, we write $r_1 \prec r_2$ iff there exist an instance K, an instance J, a homomorphism h_1 from body(r_1) to K, and a homomorphism h_2 from body(r_2) to J, such that:*

- $K \models h_2(r_2)$,
- $K \xrightarrow{r_1,h_1,\gamma_1} J$ *is a standard Chase step (for some γ_1), and*
- $J \not\models h_2(r_2)$. □

The *Chase graph* $G(\Sigma)$ of a set of dependencies Σ is a directed graph (Σ, E) containing an edge (r_1, r_2) iff $r_1 \prec r_2$. Then, Σ is *stratified* (*Str*) iff every cycle of $G(\Sigma)$ is weakly acyclic.

Intuitively, $r_1 \prec r_2$ means that firing r_1 can cause the firing of r_2.

Definition 4.4 (Stratified constraints). *[33] The* Chase graph $G(\Sigma) = (\Sigma, E)$ *of a set of constraints* Σ *contains a directed edge* (r_1, r_2) *between two constraints iff* $r_1 \prec r_2$*. We say that* Σ *is* stratified *iff the constraints in every cycle of* $G(\Sigma)$ *are weakly acyclic.* □

The class of stratified TGDs and EGDs is denoted by $\mathcal{S}tr$.

A variation of stratification, called *c-stratification*, has been proposed by [64] in order to understand whether a set of TGDs and EGDs belong to $\mathsf{CT}_{\forall}^{\mathsf{std}}$. Basically, c-stratification defines a different Chase graph and applies a constraint whenever its body is satisfied (i.e. it uses the oblivious Chase).

Definition 4.5 (C-Stratified constraints). *Given two constraints* $r_1, r_2 \in \Sigma$*, we say that* $r_1 \prec_c r_2$ *iff there exists a relational database instance* K *and two homomorphisms* h_1 *and* h_2 *such that:*

i) $K \overset{*,r_1,h_1}{\rightarrow} J$,
ii) $J \not\models h_2(r_2)$, *and*
iii) $K \models h_2(r_2)$.

The c-Chase graph $G_c(\Sigma) = (\Sigma, E)$ *of a set of dependencies* Σ *contains a directed edge* (r_1, r_2) *between two constraints iff* $r_1 \prec_c r_2$*. We say that* Σ *is c-stratified iff the dependencies in every cycle of* $G_c(\Sigma)$ *are weakly acyclic.* □

The class of c-stratified dependencies is denoted by $\mathcal{CS}tr$.

Since c-stratification guarantees the termination of all standard Chase sequences (in contrast to stratification), the class of c-stratified constraints is strictly included in the set of stratified ones ($\mathcal{CS}tr \subsetneq \mathcal{S}tr$).

4.2.4 Safety

An extension of weak acyclicity, called *safety*, which takes into account only affected positions has been proposed in [66]. An affected position denotes a position which could be associated with null values.

Definition 4.6 (Affected positions). *Let* Σ *be a set of TGDs. The set of* affected positions $aff(\Sigma)$ *of* Σ *is defined as follows. Let* R_i *be a position occurring in the head of some TGD* $r \in \Sigma$*, then*

- *if an existentially quantified variable appears in* R_i*, then* $R_i \in aff(\Sigma)$*;*
- *if the same universally quantified variable* X *appears both in position* R_i *and only in affected positions in the body of* r*, then* $R_i \in aff(\Sigma)$*.* □

Definition 4.7 (Safe set of TGDs). *[66] Let Σ be a set of TGDs, then $prop(\Sigma) = (aff(\Sigma), E)$ denotes the* propagation graph *of Σ defined as follows. For every TGD $\phi(\mathbf{X}, \mathbf{Z}) \rightarrow \exists \mathbf{Y}\, \psi(\mathbf{X}, \mathbf{Y})$ and for every X in $\mathbf{X}$ occurring in ϕ in position R_i then*

- *if $\mathbf{X}$ occurs only in affected positions in ϕ then for every occurrence of X in ψ in position S_j there is an edge $R_i \rightarrow S_j$ in E;*
- *if X occurs only in affected positions in ϕ then, for every Y in $\mathbf{Y}$ and for every occurrence of Y in ψ in position S_j there is a special edge $R_i \xrightarrow{*} S_j$ in E.*

A set of constraints Σ is said to be safe *if the corresponding propagation graph $prop(\Sigma)$ has no cycles going through a special edge.* □

The class of safe sets of TGDs is denoted by $\mathcal{SC}$.

4.2.5 Super weak acyclicity

Super-weak acyclicity [63] is a proper extension of safety (and thus weak acyclicity), and its aim is to identify sets of TGDs belonging to $\mathsf{CT}_{\forall}^{\mathsf{sobl}} = \mathsf{CT}_{\exists}^{\mathsf{sobl}}$. It builds upon a *trigger graph* $\Upsilon(\Sigma) = (\Sigma, E)$ where edges define relations among TGDs. An edge $r_i \rightsquigarrow r_j$ means that a null value introduced by a TGD r_i is propagated (directly or indirectly) into the head of r_j.

Let Σ be a set of TGDs and let $\mathcal{LP}(\Sigma)$ be the logic program obtained by replacing every TGD $\forall \mathbf{X} \forall \mathbf{Z}\, \phi(\mathbf{X}, \mathbf{Z}) \rightarrow \exists \mathbf{Y}\, \psi(\mathbf{X}, \mathbf{Y})$ with its corresponding skolemized version for the semi-oblivious Chase (see Section 2.2.2).

A *place* is a pair (a, i) where a is an atom of $\mathcal{LP}(\Sigma)$ and $0 \leq i \leq arity(a)$. Given a TGD r and an existential variable Y in the head of r, $Out(r, Y)$ denotes the set of places (called *output places*) in the head of $\mathcal{LP}(r)$ where a term of the form $f_Y^r(\mathbf{X})$ occurs. Let r be a TGD r and let X be a universal variable of r, $In(r, X)$ denotes the set of places (called *input places*) in the body of r where X occurs.

Two places (a, i) and (a', i) are *unifiable*, denoted as $(a, i) \sim (a', i)$, iff there exist two substitutions θ and θ' of (respectively) the variables a and a' such that θ, θ' map variables of a and a' to constants or slkolem terms, and $\theta(a) = \theta'(a')$. Given two sets of places Q and Q', we write $Q \sqsubseteq Q'$ iff for all $q \in Q$ there exists some $q' \in Q'$ such that $q \sim q'$.

For any set Q of places, $Move(\Sigma, Q)$ denotes the smallest set of places Q' such that $Q \subseteq Q'$, and for every constraint $r = B_r \rightarrow H_r$ in $\mathcal{LP}(\Sigma)$ and every

variable X, if $\Pi_X(B_r) \sqsubseteq Q'$ then $\Pi_X(H_r) \subseteq Q'$, where $\Pi_X(B_r)$ and $\Pi_X H_r)$ denote the sets of places in B_r and H_r where X occurs.

Definition 4.8 (Trigger graph and Super-weak Acyclicity). *[63] Given a set Σ of TGDs and two TGDs $r, r' \in \Sigma$, we say that r triggers r' in Σ, and we write $r \rightsquigarrow r'$, iff there exists an existential variable Y in the head of r, and a universal variable X' occurring both in the body and head of r' such that $In(r', X') \sqsubseteq Move(\Sigma, Out(r, Y))$. A set of constraints Σ is super-weakly acyclic iff the trigger graph $\Upsilon(\Sigma) = (\Sigma, \{(r_1, r_2) | r_1 \rightsquigarrow r_2\})$ is acyclic.* □

The class of super-weakly acyclic sets of TGDs is denoted by $\mathcal{S}w\mathcal{A}$.

4.2.6 Safe restriction and Inductive restriction

A more refined extension of both c-stratification and safety has been proposed under the name of *safe restriction* [66, 64], working on TGDs and EGDs. Basically, safe restriction refines stratification by considering constraint firing and possible propagation of null values together.

In order to introduce this concept we need some further definitions. For any set of positions P and a TGD r, $aff(r, P)$ denotes the set of positions π from the head of r such that i) for every universally quantified variable X in π, X occurs in the body of r only in positions from P or ii) π contains an existentially quantified variable.

For any $r_1, r_2 \in \Sigma$ and $P \subseteq pos(\Sigma)$, $r_1 \prec_P r_2$ if

1. $r_1 \prec_c r_2$ (i.e. there exists an instance K and two homomorphisms h_1 and h_2 such that i) $K \stackrel{r_1, h_1}{\rightarrow} J$, ii) $J \not\models h_2(r_2)$ and iii) $K \models h_2(r_2)$), and
2. there is null value propagated from the body to the head of $h_2(r_2)$ such that it occurs in K only in positions from P.

Definition 4.9 (Safe restriction). *A 2-restriction system is a pair $(G'(\Sigma), P)$, where $G'(\Sigma) = (\Sigma, E)$ is a directed graph and $P \subseteq pos(\Sigma)$ such that:*

1. *for all $(r_1, r_2) \in E$: if r_1 is a TGD, then $aff(r_1, P) \cap pos(\Sigma) \subseteq P$, whereas if r_2 is a TGD, then $aff(r_2, P) \cap pos(\Sigma) \subseteq P$, and*
2. *$r_1 \prec_P r_2 \ \Rightarrow \ (r_1, r_2) \in E$.*

Σ is called safely restricted *if and only if there is a restriction system $(G'(\Sigma), P)$ for Σ such that every strongly connected component in $G'(\Sigma)$ is safe.* □

A 2-restriction system is *minimal* if it is obtained from $((\Sigma, \varnothing), \varnothing)$ by a repeated application of conditions 1 and 2 of Definition 4.9 (until both conditions hold considering all constraints) such that, in case Condition 1 is applied, P is extended only by those positions that are required to satisfy the condition. [65] has shown that Σ is safely restricted if and only if every strongly connected component in $G'(\Sigma)$ is safe, where $(G'(\Sigma), P)$ is the minimal 2-restriction system for Σ.

Safely restriction has been further extended into a criterion called *inductive restriction*, whose main idea is to decompose a given set of TGDs and EGDs into smaller subsets (in a more refined way than safe restriction). In particular, $\mathcal{IR}$ first computes the system $(G'(\Sigma), P)$ and partitions Σ into $\Sigma_1, ..., \Sigma_n$, where each Σ_i is a set of dependencies defining a strongly connected component in $G'(\Sigma)$, next, if $n = 1$ the safety criterion is applied to Σ, otherwise the inductive restriction criterion is applied inductively to each Σ_i. The class of inductive restricted dependencies is denoted by $\mathcal{IR}$.

The problem of checking whether a set of dependencies is inductively restricted is in co-NP. As well as c-stratification and safety, inductive restriction guarantees that for every database D there exists a polynomial in the size of D that bounds the length of every Chase sequence of D with Σ [64]

Inductive restriction has been further extended by considering not only the relationships among pairs of constraints, but general sequences of m constraints, with $m \geq 2$ [66]. The use of sequences of $m \geq 2$ constraints allows a hierarchy of classes where each class is characterized by m and denoted by $\mathcal{T}[m]$, with $\mathcal{T}[2] = \mathcal{IR}$ and $\mathcal{T}[m] \subsetneq \mathcal{T}[m+1]$.

4.2.7 Local Stratification

In [55], an extension of both $\mathcal{IR}$ and $\mathcal{S}w\mathcal{A}$ was proposed. We start by introducing a notion of *fireable place*. We say that a place q appearing in the body of a dependency r could be fired by a place q' appearing in the head of constraint r', denoted by $q' < q$, if $q \sim q'$ and $r' < r$. Given two sets of places Q and Q', we say that Q could be fired by Q', denoted by $Q' < Q$, iff for all $q \in Q$ there exists some $q' \in Q'$ such that $q' < q$.

Given a set Q of places, we define $MOVE(\Sigma, Q)$ as the smallest set of places Q' such that: i) $Q \subseteq Q'$, and ii) for every constraint $r = B_r \to H_r$ in $sk(\Sigma)$ and every variable X, if $Q' < \Pi_X(B_r)$, then $\Pi_X(H_r) \subseteq Q'$. Here $\Pi_X(B_r)$ and $\Pi_X(H_r)$ denote the sets of places in B_r and H_r where X occurs.

With respect to the function $Move$, the new function $MOVE$ here considered takes into account the firing of places and not only the unification of places.

Definition 4.10 (Local Stratification). *Given a set Σ of TGDs and two TGDs $r_1, r_2 \in \Sigma$, we say that r_1 triggers r_2 in Σ and write $r_1 \hookrightarrow r_2$ iff there exists an existential variable Y in the head of r_1, and a universal variable X occurring both in the body and head of r_2 such that $MOVE(\Sigma, Out(r_1, Y)) < In(r_2, x)$. A set of constraints Σ is* locally stratified *iff the trigger graph $\Delta(\Sigma) = \{(r_1, r_2) | r_1 \hookrightarrow r_2\}$ is acyclic.* □

The class of locally stratified constraints is denoted by $\mathcal{LS}$.

4.2.8 Model-faithful acyclicity

In [50], the classes of *model-faithful acyclic* ($\mathcal{MFA}$) and *model-summarising acyclic* ($\mathcal{MSA}$) TGDs have been proposed. The idea is to run the oblivious (or semi-oblivious) Chase and then use sufficient checks to identify cyclic computations. Since no sufficient, necessary, and computable test can be given for the latter, [50] adopted an approach of "raising the alarm" and stop the process if a "cyclic" term $f(t)$ is derived, i.e., where f occurs in t. This is done in a declarative way by extending a given set of dependencies Σ into a new set Σ', and then checking whether Σ' does not entail a special predicate. The two aforementioned techniques are defined for TGDs only, as EGDs are assumed to be emulated through substitution-free simulation (discussed in Section 4.4).

4.2.9 Rewriting technique

Rewriting techniques for checking Chase termination have been proposed in [54, 55, 56]. They consist in rewriting a set of TGDs Σ into a new set Σ^α with the aim of verifying structural properties for Chase termination on Σ^α rather than Σ. These techniques have been defined for TGDs only and perform an analysis of the semi-oblivious Chase. [56] showed that most of the termination criteria improve if we consider rewritten TGDs rather than the original ones. The rewriting approach has also been used to define the class $\mathcal{AC}$ of acyclic TGDs.

4.3 Termination of Guarded rules

Although the Chase termination problem is undecidable in general, the proof given in [47] does not show the undecidability of the problem for TGDs that enjoy some structural conditions, which in turn guarantee favorable model-theoretic properties. Such a key condition is *guardedness*, a well-accepted paradigm that gives rise to robust rule-based languages [11, 22, 23] that capture important databases constraints such as inclusion dependencies, and lightweight description logics such as DL-Lite [27] and $\mathcal{EL}$ [10].

Guardedness guarantees the tree-likeness of the underlying models, and thus the decidability of central database problems such as query answering and containment under constraints. The question that comes up is whether guardedness has the same positive impact on the Chase termination problem:

Question 1: Given a set Σ of guarded TGDs, is it possible to decide whether, for every database D, the Chase on D and Σ terminates?

Of course, if the answer to the above question is positive, then the next step is to understand how complex is the problem of determining whether the Chase terminates:

Question 2: Given a set Σ of guarded TGDs, what is the exact complexity of deciding whether, for every database D, the Chase on D and Σ terminates?

Our main goal in this section is to study in depth the Chase termination problem for guarded TGDs, and give answers to the above fundamental questions. In fact, we focus on the (semi-)oblivious versions of the Chase, and we show that deciding termination for guarded TGDs is decidable [19]. This is the first contribution that establishes positive results for the (semi-)oblivious Chase termination problem.

4.3.1 Linearity

We proceed to investigate the (semi-)oblivious Chase termination problem for (simple) linear TGDs. The goal of this section is twofold: for every $\star \in \{\mathsf{obl}, \mathsf{sobl}\}$,

1. Syntactically characterize the classes $(\mathsf{CT}^\star \cap \mathsf{SL})$ and $(\mathsf{CT}^\star \cap \mathsf{L})$; and
2. Pinpoint the complexity of the $\star$-chase termination problem for sets of TGDs of $(\mathsf{S})\mathsf{L}$.

For our first goal, we are going to exploit existing syntactic conditions that guarantee the termination of every (semi-)oblivious Chase sequence on all databases; in fact, our analysis will build on rich-acyclicity [58] and weak-acyclicity [38]. More precisely, we are going to show that for simple linear TGDs rich-acyclicity (resp., weak-acyclicity) is enough for characterizing $(\mathsf{CT}^{\mathsf{obl}} \cap \mathsf{SL})$ (resp., $(\mathsf{CT}^{\mathsf{sobl}} \cap \mathsf{SL})$). However, for (non-simple) linear TGDs this is not the case, and we need to carefully extend rich- and weak-acyclicity. The above syntactic characterizations, apart from being interesting in their own right, allow us to obtain optimal upper bounds for the $\star$-chase termination problem for $(\mathsf{S})\mathsf{L}$, and thus achieving our second goal — we simply need to analyze the complexity of deciding whether a set of (simple) linear TGDs enjoys the above acyclicity-based conditions.

In the sequel, we assume a fixed order on the head-atoms of TGDs.

Characterizing $(\mathsf{CT}^{\mathsf{obl}} \cap \mathsf{SL})$ and $(\mathsf{CT}^{\mathsf{sobl}} \cap \mathsf{SL})$

Oblivious Chase

We start our investigation by showing that rich-acyclicity characterizes the fragment of SL that guarantees the termination of the oblivious Chase. In particular, we prove that:

Theorem 4.11. $(\mathsf{CT}^{\mathsf{obl}} \cap \mathsf{SL}) \ = \ (\mathsf{RA} \cap \mathsf{SL})$. □

To establish the above theorem it suffices to show that, for an arbitrary set of TGDs $\Sigma \in \mathsf{SL}$, $\Sigma \in \mathsf{CT}^{\mathsf{obl}}$ iff $\Sigma \in \mathsf{RA}$. The "if" direction has been shown in [58]. Assume now that $\Sigma \notin \mathsf{RA}$. We are going to show that there exists a database D, and a non-terminating obl-chase sequence of D w.r.t. Σ, which immediately implies that $\Sigma \notin \mathsf{CT}^{\mathsf{obl}}$. But let us first introduce our generic technical tool, which will be used also for the semi-oblivious Chase, and all the other languages that we treat in this work. Given a TGD r, $\diamond_r^\star$ is defined as $\neq$, if $\star = \mathsf{obl}$, and $\not\sim_r$, if $\star = \mathsf{sobl}$.

Definition 4.12. *We say that a set Σ of TGDs admits an* infinite $\star$-chase derivation*, where $\star \in \{\mathsf{obl}, \mathsf{sobl}\}$, if there exist infinite sequences $I_0, I_1, \ldots$ and $(r_0, h_0), (r_1, h_1), \ldots$, where $r_0, r_1, \ldots \in \Sigma$, such that*

1. *for each $i \geqslant 0$, $I_i \xrightarrow{r_i, h_i} I_{i+1}$; and*
2. *for each $i \neq j \geqslant 0$, $r_i = r_j = r$ implies $h_i \diamond_r^\star h_j$.* □

It is possible to show that the $\star$-chase termination problem, where $\star \in \{\mathsf{obl}, \mathsf{sobl}\}$, is tantamount to the problem of deciding whether a set of TGDs admits an infinite $\star$-chase derivation.

Proposition 4.13. *Consider a set Σ of TGDs. $\Sigma \notin \mathsf{CT}^{\star}$ iff Σ admits an infinite $\star$-chase derivation, where $\star \in \{\mathsf{obl}, \mathsf{sobl}\}$.* □

The "only-if" direction is trivial. For the "if" direction, it suffices to show that there exists a database D, and a non-terminating $\star$-chase sequence of D w.r.t. Σ. By hypothesis, we have sequences $I_0, I_1, \ldots$ and $(r_0, h_0), (r_1, h_1), \ldots$ as in Definition 4.12. A non-terminating $\star$-chase sequence of I_0 w.r.t. Σ is

$$J_0, J_0^1, \ldots, J_0^{k_0}, J_1, J_1^1, \ldots, J_1^{k_1}, J_2, \ldots$$

where,

- $J_0 = I_0$;
- for each $i \geqslant 0$, there exists a trigger (r, h) for Σ on J_i such that $J_i\langle r, h\rangle J_i^1$;
- for each $i \geqslant 0$ and $1 \leqslant j < k_i$, there exists a trigger (r, h) for Σ on J_i such that $J_i^j\langle r, h\rangle J_i^{j+1}$;
- for each $i \geqslant 0$, $J_i^{k_i}\langle r_i, h_i\rangle J_{i+1}$; recall that (r_i, h_i) is a trigger occurring in the sequence obtained by hypothesis;
- for each pair of triggers (r, h) and (r', h') considered above, $r = r'$ implies $h \diamond_r^{\star} h'$; and
- for each $i \geqslant 0$, $k_i \geqslant 0$ is the maximal integer such that the above conditions hold.

Intuitively speaking, the above Chase sequence constructs the Chase in a level-by-level fashion, where level zero is defined as J_0, and the atoms of level i are obtained by applying TGDs on atoms of level $i - 1$, by giving priority to the triggers $(r_0, h_0), (r_1, h_1), \ldots$. Therefore, Proposition 4.13 follows.

We proceed now with the proof of Theorem 4.11. Recall that we need to show the following: for the set $\Sigma \in \mathsf{SL}$, $\Sigma \notin \mathsf{RA}$ implies $\Sigma \notin \mathsf{CT}^{\mathsf{obl}}$. By Proposition 4.13, it suffices to show that, if $\Sigma \notin \mathsf{RA}$, then Σ admits an infinite obl-chase derivation. The rest of this section is devoted to establish that indeed Σ admits an infinite obl-chase derivation.

By hypothesis, there exists a cycle in $EDG(\Sigma)$ that contains a special edge; let $(v_0, v_1), (v_1, v_2), \ldots, (v_{n-1}, v_n)$ be such a cycle ($v_0 = v_n$) with $\lambda((v_i, v_{i+1})) = (r_i, k_i)$, for each $0 \leqslant i < n$. In the sequel, we refer to the above cycle by C. One may claim that, starting from a database D that triggers the

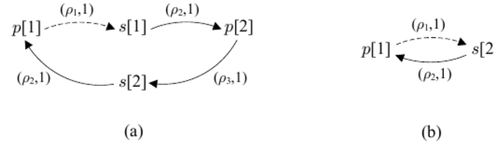


Fig. 4.2: Cycles for Examples 4.14 and 4.15.

TGD r_0, the cycle C will give rise to an infinite obl-chase derivation, which in turn implies that Σ admits an infinite obl-chase derivation, as needed. However, such a derivation may be invalid due to the fact that the involved triggers are not distinct. In other words, there is no guarantee that the edges of C that are labeled with the same TGD give rise to different triggers.

Example 4.14. Consider the set $\Sigma' \in \mathsf{SL}$ consisting of

$$\begin{aligned}\rho_1 &= p(X,Y) \rightarrow \exists Z\, s(Z,Z)\\ \rho_2 &= s(X,Y) \rightarrow p(Y,X)\\ \rho_3 &= p(X,Y) \rightarrow s(X,Y).\end{aligned}$$

It is easy to verify that the cycle depicted in Figure 4.2(a), where the dashed arrow represents a special edge, occurs in $EDG(\Sigma')$. Starting from $I_0 = \{p(c,c)\}$, where $c \in \mathbf{C}$, if we apply the TGDs as dictated by this cycle, we get an infinite sequence of instances $I_0, I_1, \ldots$ with

$$\begin{aligned}I_1 &= I_0 \cup \{s(z_1,z_1)\}\\ I_2 &= I_3 = I_4 = I_1 \cup \{p(z_1,z_1)\}\\ I_5 &= I_4 \cup \{s(z_2,z_2)\}\\ I_6 &= I_7 = I_8 = I_5 \cup \{p(z_2,z_2)\}\\ &\ldots\end{aligned}$$

where $z_1, z_2, \ldots$ are nulls. However, this sequence is not a valid obl-chase derivation since, for each $i \in \{1,5,9,13,\ldots\}$, assuming that $I_i \overset{\rho_2,h}{\rightarrow} I_{i+1}$ and $I_{i+2} \overset{\rho_2,h'}{\rightarrow} I_{i+3}$, $h = h' = \{X_2 \rightarrow z_{\lceil \frac{i}{4} \rceil}, Y_2 \rightarrow z_{\lceil \frac{i}{4} \rceil}\}$. Thus, (ρ_2,h), (ρ_2,h') are not distinct, as required by an infinite obl-chase derivation. □

Although C does not necessarily encode a valid infinite obl-chase derivation, it is possible to show that in $EDG(\Sigma)$ there exists a cycle C', whose length is less or equal than the length of C, which encodes a valid infinite obl-chase derivation. Intuitively speaking, if we avoid to reapply the repeated triggers that are involved in the infinite sequence of instances obtained due to

C, then we get a valid obl-chase derivation, which corresponds to C'. In fact, C' is one of the shortest cycles in $EDG(\Sigma)$ that contains a special edge. Let us illustrate this via an example that builds on Example 4.14.

Example 4.15. Consider the set Σ' given in Example 4.14. As already discussed above, starting from $I_0 = \{p(c, c)\}$, and applying the TGDs as dictated by the cycle of $EDG(\Sigma)$ shown in Figure 4.2(a), we obtain an infinite sequence of instances that is not a valid obl-chase derivation, since some of the involved triggers are repeated. If we avoid to reapply those triggers, then we get an infinite sequence of instances $J_0 = I_0, J_1, \ldots$ with

$$\begin{array}{lll} J_1 = J_0 \cup \{s(z_1, z_1)\} & & J_2 = J_1 \cup \{p(z_1, z_1)\} \\ J_3 = J_2 \cup \{s(z_2, z_2)\} & & J_4 = J_3 \cup \{p(z_2, z_2)\} \\ \ldots & & \end{array}$$

where $z_1, z_2, \ldots$ are nulls of $\mathbf{N}$. It is easy to verify that $J_0, J_1, \ldots$ is a valid infinite obl-chase derivation, and that this derivation corresponds to the cycle of $EDG(\Sigma')$ depicted in Figure 4.2(b). This cycle is of length two, and there is no shorter cycle that contains a special edge. □

From the above discussion, one can exploit the minimal cycles in the extended dependency graph, and show that:

Lemma 4.16. *For every set $\Sigma \in \mathsf{SL}$, if $\Sigma \notin \mathsf{RA}$, then Σ admits an infinite* obl*-chase derivation.* □

By Proposition 4.13 and Lemma 4.16, we immediately get that $\Sigma \notin \mathsf{CT}^{\mathsf{obl}}$, and Theorem 4.11 follows.

Semi-Oblivious Chase

By following a similar approach, we can characterize the fragment of SL that guarantees the termination of the semi-oblivious Chase. Clearly, for a set of TGDs Σ, $\Sigma \in (\mathsf{RA} \cap \mathsf{SL})$ implies $\Sigma \in (\mathsf{CT}^{\mathsf{obl}} \cap \mathsf{SL})$, which in turn implies $\Sigma \in (\mathsf{CT}^{\mathsf{sobl}} \cap \mathsf{SL})$. However, the other direction is, in general, not true. Consider the set Σ given in Example 4.2. It is easy to verify that $\Sigma \in (\mathsf{CT}^{\mathsf{sobl}} \cap \mathsf{SL})$, but $\Sigma \notin (\mathsf{RA} \cap \mathsf{SL})$, since in its extended dependency graph, which is depicted in Figure 4.1, there exists a cycle that contains a special edge.

The main reason why rich-acyclicity is not enough for characterizing $(\mathsf{CT}^{\mathsf{sobl}} \cap \mathsf{SL})$, is the existence (in the extended dependency graph) of the

special edges from the positions where non-frontier variables occur to the positions where existentially quantified variables appear. In fact, those edges encode erroneous propagations of nulls that do not take place during the construction of the semi-oblivious Chase. Recall that after eliminating those problematic special edges, we get a graph structure that coincides with the dependency graph. This observation led us to conjecture that weak-acyclicity is enough for characterizing $(\mathsf{CT}^{\mathsf{sobl}} \cap \mathsf{SL})$. By giving a proof similar to that of Lemma 4.16, with the difference that we exploit the dependency graph instead of the extended dependency graph, we show that:

Lemma 4.17. *For every set $\Sigma \in \mathsf{SL}$, if $\Sigma \notin \mathsf{WA}$, then Σ admits an infinite* sobl*-chase derivation.* □

By Proposition 4.13 and Lemma 4.17, we immediately get that $\Sigma \notin \mathsf{WA}$ implies $\Sigma \notin \mathsf{CT}^{\mathsf{sobl}}$. Notice that the other direction is implicit in [63], where the same has been shown for a superclass of WA, and the next result follows:

Theorem 4.18. $(\mathsf{CT}^{\mathsf{sobl}} \cap \mathsf{SL}) \ = \ (\mathsf{WA} \cap \mathsf{SL})$. □

Consequences to Other Formalisms

Despite their simplicity, simple linear TGDs are powerful enough for capturing prominent database dependencies, and in particular *inclusion dependencies*; see, e.g., [5]. It is well-known that inclusion dependencies correspond to simple linear (constant-free) TGDs with just one head-atom without repetition of variables, and we refer to this formalism by ID. Furthermore, simple linear TGDs generalize prominent ontology languages, and in particular DL-Lite$_{\mathcal{R}}$ [27]. In fact, DL-Lite$_{\mathcal{R}}$ (ignoring disjointness and non-membership axioms) corresponds to simple linear (constant-free) TGDs that use only unary and binary predicates; we refer to this formalism by $\mathsf{DL\text{-}Lite}^{\mathsf{TGD}}$. It is evident that our preceding results on simple linear TGDs immediately imply the following:

Corollary 4.19. *It holds that,*

1. $(\mathsf{CT}^{\star} \cap \mathsf{ID}) \ = \ (\mathit{L}(\star) \cap \mathsf{ID})$,
2. $(\mathsf{CT}^{\star} \cap \mathsf{DL\text{-}Lite}^{\mathsf{TGD}}) \ = \ (\mathit{L}(\star) \cap \mathsf{DL\text{-}Lite}^{\mathsf{TGD}})$,

where $\star \in \{\mathsf{obl}, \mathsf{sobl}\}$, $\mathit{L}(\mathsf{obl}) = \mathsf{RA}$, *and* $\mathit{L}(\mathsf{sobl}) = \mathsf{WA}$. □

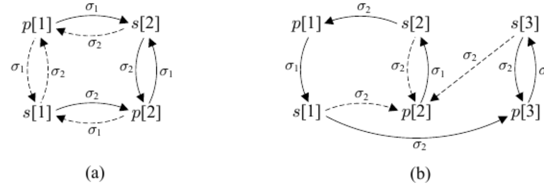


Fig. 4.3: Extended dependency graphs of Examples 4.20 and 4.23.

Characterizing ($\mathsf{CT}^{\mathsf{obl}} \cap \mathsf{L}$) and ($\mathsf{CT}^{\mathsf{sobl}} \cap \mathsf{L}$)

Oblivious Chase

We proceed with the characterization of the fragment of L that guarantees the termination of the oblivious Chase. Let us first expose, by means of simple examples, the two reasons for which rich-acyclicity is not enough for our purposes.

Example 4.20. Consider the set $\Sigma \in \mathsf{L}$ consisting of

$$r_1 = p(X,X) \to \exists Z\, s(Z,X)$$
$$r_2 = s(X,X) \to \exists Z\, p(Z,X).$$

It is easy to verify that in the extended dependency graph of Σ, depicted in Figure 4.3(a), there exists a cycle that contains a special edge. However, for every database D, every obl-chase sequence of D w.r.t. Σ is terminating. □

As shown by the above example, the first reason why rich-acyclicity is not enough for characterizing ($\mathsf{CT}^{\mathsf{obl}} \cap \mathsf{L}$), is the fact that a cycle in the extended dependency graph does not necessarily encode a Chase derivation. Consider, for example, the cycle $(p[1], s[1]), (s[1], p[1])$, where the first edge is labeled by r_1, and the second edge by r_2. One expects that, after applying r_1 during the Chase, the obtained atom A may trigger r_2. However, this is not the case, since the atom A is necessarily of the form $s(t,t')$, where $t \neq t'$, which means that there is no homomorphism from $body(r_2)$ to A. The atom A is of the above form since, in the head-atom of r_1, at position $s[1]$ we have an existentially quantified variable, while at position $s[2]$ a frontier variable.

The above informal discussion, demonstrates the need of finding an effective way for guaranteeing that a cycle in the extended dependency graph can indeed be traversed during the construction of the Chase, in which case is called active. To this end, we need to understand when, for two single-head linear TGDs r_1 and r_2, the atom obtained by applying r_1 may trigger r_2.

Notice that we focus on single-head TGDs, since, by definition, the edges of the extended dependency graph are labeled by single-head TGDs. The above property is captured by the notion of compatibility. In the sequel, we assume the reader is familiar with the notion of unification. Given two atoms A and B that unify, we denote by $\mathsf{MGU}(A, B)$ their most general unifier.

Definition 4.21. *Let r_1 and r_2 be single-head linear TGDs. Then, r_1 is* compatible *with r_2 if: $head(r_1)$ and $body(r_2)$ unify, and for each $X \in var(body(r_2))$, assuming that $\Pi = pos(body(r_2), \{X\})$, either $var(head(r_1), \Pi) \subseteq fr(r_1)$, or, $var(head(r_1), \Pi) = \{Z\}$, for some $Z \in ex(r_1,)$.* □

Notice that, in Example 4.20, r_1 is not compatible with r_2, and vice-versa. Having the notion of compatibility in place, one may be tempted to claim that a sequence of single-head linear TGDs $r_1, \ldots, r_n$ is active if, for each $i \in [n-1]$, r_i is compatible with r_{i+1}. However, this does not capture our intention. Instead, we need to ensure that the resolvent of such a sequence, which is actually a single-head linear TGD that simulates the behavior of $r_1, \ldots, r_n$, exists.

Definition 4.22. *The* resolvent *of a sequence $r_1, \ldots, r_n$ of single-head linear TGDs, denoted $\mathsf{R}(r_1, \ldots, r_n)$, is inductively defined as follows (for convenience, we write ρ for $\mathsf{R}(r_1, \ldots, r_{n-1})$):*

1. *$\mathsf{R}(r_1) = r_1$; and*
2. *$\mathsf{R}(r_1, \ldots, r_n) = \theta(body(\rho)) \rightarrow \theta(head(r_n))$, where $\theta = \mathsf{MGU}(head(\rho), body(r_n))$ if $\rho \neq \bot$ and ρ is compatible with r_n; otherwise, $\mathsf{R}(r_1, \ldots, r_n) = \bot$.*

The sequence $r_1, \ldots, r_n$ is active *if $\mathsf{R}(r_1, \ldots, r_n) \neq \bot$.* □

Apparently, in order to achieve our goal, we need to extend rich-acyclicity to active-rich-acyclicity, by allowing cycles with special edges to appear in the extended dependency graph, as long as they do not give rise to active sequences of single-head linear TGDs. Unfortunately, active-rich-acyclicity is still not expressive enough for characterizing the fragment of linear TGDs that guarantees the termination of the oblivious Chase.

Example 4.23. Consider the set $\Sigma \in \mathsf{L}$ consisting of

$$r_1 = p(X, Y, Z) \rightarrow s(X, Y, Z)$$
$$r_2 = s(X, Y, X) \rightarrow \exists Z\, p(Y, Z, X).$$

It is easy to verify that in $EDG(\Sigma)$, depicted in Figure 4.3(b), there exists an active cycle that contains a special edge. For example, $C = (p[2], s[2]), (s[2], p[2])$ gives rise to the sequence r_1, r_2. Since r_1 is compatible with r_2, we get that $\mathsf{R}(r_1, r_2) \neq \bot$, which in turn implies that C is active. Even if Σ is not actively-richly-acyclic, we can show that, for every D, every obl-chase sequence of D w.r.t. Σ is terminating. To this aim, it suffices to verify that every obl-chase sequence of the critical database $D_{\mathsf{c}}(\{p, s\})$ w.r.t. Σ is terminating. □

The above example exposes the second reason why rich-acyclicity is not expressive enough for characterizing $(\mathsf{CT}^{\mathsf{obl}}_{\forall} \cap \mathsf{L})$. In particular, even if a cycle in the extended dependency graph is active, which means that it can be traversed at least once during the construction of the Chase, it is not guaranteed that it can be traversed infinitely many times, and thus give rise to an infinite Chase derivation. Consider, for example, the cycle $C = (p[2], s[2]), (s[2], p[2])$, where the first edge is labeled by r_1, and the second edge by r_2. Since C is active, one expects that, starting from $p(\mathsf{c}, \mathsf{c}, \mathsf{c})$, where p is the predicate of $\mathit{body}(r_1)$, we can apply $r_1, r_2, r_1, \ldots$ infinitely many times during the Chase. However, after applying r_1, r_2, r_1, r_2, r_1, we obtain an atom $A = s(t, t', \mathsf{c})$, where $t \neq t'$, and thus there is no homomorphism from $\mathit{body}(r_2)$ to A. In other words, the cycle C can be traversed twice, but during its third traversal r_2 is not triggered. The reason for this behavior is the fact that the sequence $\mathsf{R}(r_1, r_2), \mathsf{R}(r_1, r_2), \mathsf{R}(r_1, r_2)$ of length *three* — recall that the Chase derivation is blocked during the *third* traversal of C — is not active.

It is clear that we need an effective way for ensuring that an active cycle in the extended dependency graph can be traversed infinitely many times during the construction of the Chase. In particular, assuming that an active cycle is labeled by the TGDs $r_1, \ldots, r_n$, we need to ensure that, for every $k > 0$, if $\rho = \mathsf{R}(r_1, \ldots, r_n)$, the sequence $\rho, \ldots, \rho$ of length k is active, in which case $r_1, \ldots, r_n$ is critical. Interestingly, as we shall see below, for ensuring the above criticality condition, we only need to consider sequences of length up to $(\omega_{r_1} + 1)$, where ω_{r_1} is the arity of the predicate of $\mathit{body}(r_1)$. This leads to the following definition of critical sequences. Henceforth, r^k denotes the sequence $r, \ldots, r$ of length k:

Definition 4.24. *A sequence $r_1, \ldots, r_n$ of single-head linear TGDs is* critical *if:*

1. *It is active; and*

2. *For each* $k \in [\omega_{r_1} + 1]$*, the sequence* ρ^k*, where* $\rho = \mathsf{R}(r_1, \ldots, r_n)$*, is active.* □

We are now ready to extend rich-acyclicity to critical-rich-acyclicity, which, as we shall see, is the formalism that characterizes $(\mathsf{CT}^{\mathsf{obl}} \cap \mathsf{L})$.

Definition 4.25. *Consider a set* $\Sigma \in \mathsf{L}$*, and let* $EDG(\Sigma) = (N, E, \lambda)$*. A cycle* $(v_0, v_1), \ldots, (v_{n-1}, v_n)$ *in* $EDG(\Sigma)$ *is called* critical*, if* $\lambda((v_0, v_1)), \ldots, \lambda((v_{n-1}, v_n))$ *is critical. We say that* Σ *is* critically-richly-acyclic*, if no critical cycle in* $EDG(\Sigma)$ *contains a special edge, and the corresponding class is denoted* $\mathsf{LCriticalRA}$*.* □

The main result of this section follows:

Theorem 4.26. $(\mathsf{CT}^{\mathsf{obl}} \cap \mathsf{L}) = \mathsf{LCriticalRA}$*.* □

The "if" direction of the above result is shown by giving a proof similar to the one given in [58] for showing that $\Sigma \in \mathsf{RA}$ implies $\Sigma \in \mathsf{CT}^{\mathsf{obl}}$. The interesting part is to show that, for a set $\Sigma \in \mathsf{L}$, $\Sigma \notin \mathsf{LCriticalRA}$ implies $\Sigma \notin \mathsf{CT}^{\mathsf{obl}}$. By Proposition 4.13, it suffices to show that, if $\Sigma \notin \mathsf{LCriticalRA}$, then Σ admits an infinite obl-chase derivation. This is a rather non-trivial task, which requires some intermediate results.

The equality type of an atom is a set of equalities among positions that perfectly describe its shape. Formally, given a (constant-free) atom $A = p(X_1, \ldots, X_n)$, the *equality type* of A is defined as the set $\mathit{eqtype}(A) = \{p[i] = p[j] \mid X_i = X_j\}$. For a linear TGD r, let $\mathit{eqtype}(r) = \mathit{eqtype}(\mathit{body}(r))$. The following lemma establishes a useful property about active sequences and equality types:

Lemma 4.27. *Let* r *be a single-head linear TGD such that* r^i *and* r^{i+1} *are active, for some integer* $i > 0$*, and* $\mathit{eqtype}(\mathsf{R}(r^i)) = \mathit{eqtype}(\mathsf{R}(r^{i+1}))$*. Then,* r^{i+2} *is active, and* $\mathit{eqtype}(\mathsf{R}(r^{i+1})) = \mathit{eqtype}(\mathsf{R}(r^{i+2}))$*.* □

The above result allows us to show that critical cycles can be traversed infinitely many times during the construction of the Chase, starting from the critical database.

Consider a critical sequence $r_1, \ldots, r_n$ of single-head linear TGDs, and let $\rho = \mathsf{R}(r_1, \ldots, r_n)$. It is not difficult to show that there exists $i \in [\omega_{r_1} + 1]$ such that ρ^i and ρ^{i+1} are active, and $\mathit{eqtype}(\mathsf{R}(\rho^i)) = \mathit{eqtype}(\mathsf{R}(\rho^{i+1}))$. By recursively applying Lemma 4.27, we conclude that, for every $k > [\omega_{r_1} + 1]$, ρ^k is active. Moreover, since $r_1, \ldots, r_n$ is critical, for every $k \in [\omega_{r_1} + 1]$, ρ^k is active. From the above discussion, we get the following crucial result:

Lemma 4.28. *Let $r_1, \dots, r_n$ be a critical sequence of single-head linear TGDs. Then, for every $k > 0$, ρ^k, where $\rho = \mathsf{R}(r_1, \dots, r_n)$, is active.* □

By using Lemma 4.28, and the fact that the resolvent of an active sequence of single-head linear TGDs mimics the behavior of the sequence during the Chase, starting from the critical database (this can be easily shown by induction on the length of the sequence), we can establish that a minimal critical cycle that contains a special edge gives rise to an infinite Chase derivation, which in turn implies the following:

Lemma 4.29. *For every set $\Sigma \in \mathsf{L}$, if $\Sigma \notin \mathsf{LCriticalRA}$, then Σ admits an infinite* obl*-chase derivation.* □

By Proposition 4.13 and Lemma 4.29, we get that $\Sigma \notin \mathsf{LCriticalRA}$ implies $\Sigma \notin \mathsf{CT}^{\mathsf{obl}}$, and Theorem 4.26 follows.

Semi-Oblivious Chase

By applying similar techniques, we can characterize the fragment of L that guarantees the termination of the semi-oblivious Chase. Towards this direction, we first need to introduce the notion of critical-weak-acyclicity, which is defined as critical-rich-acyclicity, with the difference that the desired condition is posed on the dependency graph, and not on the extended dependency graph.

Definition 4.30. *A set $\Sigma \in \mathsf{L}$ is* critically-weakly-acyclic*, if no critical cycle in $DG(\Sigma)$ contains a special edge, and the corresponding class is denoted* $\mathsf{LCriticalWA}$. □

As already discussed in Section 4.3.1, the extended dependency graph encodes propagations of nulls that do not take place during the construction of the semi-oblivious Chase, and this is exactly the reason why we need to rely on the dependency graph for the characterization of $(\mathsf{CT}^{\mathsf{sobl}} \cap \mathsf{L})$. By giving a proof similar to that of Lemma 4.29, with the difference that we exploit the dependency graph instead of the extended dependency graph, we show that:

Lemma 4.31. *For every set $\Sigma \in \mathsf{L}$, if $\Sigma \notin \mathsf{LCriticalWA}$, then Σ admits an infinite* sobl*-chase derivation.* □

By Proposition 4.13 and Lemma 4.31, $\Sigma \notin \mathsf{LCriticalWA}$ implies $\Sigma \notin \mathsf{CT}^{\mathsf{sobl}}$. The proof of the other direction is along the lines of the proof given in [38] for showing that weak-acyclicity guarantees the termination of the standard Chase, and we get that:

Theorem 4.32. $(\mathsf{CT}^{\mathsf{sobl}} \cap \mathsf{L}) = \mathsf{LCriticalWA}$. □

Complexity

Let us now proceed with our second goal, that is, to pinpoint the complexity of the $\star$-chase termination problem for sets of TGDs of $(\mathsf{S})\mathsf{L}$, where $\star \in \{\mathsf{obl}, \mathsf{sobl}\}$.

Simple Linear TGDs

We first focus on simple linear TGDs, and we show the following:

Theorem 4.33. *Consider a set $\Sigma \in \mathsf{SL}$. The problem of deciding whether $\Sigma \in \mathsf{CT}^{\star}$, where $\star \in \{\mathsf{obl}, \mathsf{sobl}\}$, is* NL*-complete, even for unary and binary predicates.* □

Upper Bound. To obtain the upper bound, by Theorems 4.11 and 4.18, it suffices to show that deciding whether Σ is richly-acyclic (or weakly-acyclic) is in NL.

Lemma 4.34. *Consider $\Sigma \in \mathsf{SL}$. The problem of deciding if $\Sigma \in L$, where $L \in \{\mathsf{RA}, \mathsf{WA}\}$, is in $NSPACE(\log(\omega \cdot |sch(\Sigma)|))$, where ω is the maximum arity of $sch(\Sigma)$.* □

The complement of the problem under consideration can be seen as an instance of graph reachability. In fact, we need to decide whether there exists a node v in the (extended) dependency graph of Σ that is reachable from itself, with the additional condition that the path from v to itself contains at least one special edge. This can be done via a nondeterministic procedure, where at each step needs to remember two consecutive edges of the graph (i.e., three positions of $sch(\Sigma)$), the origin of the traversed cycle (i.e, the position v), and a binary value indicating whether a special edge has been visited or not. All the above elements can be maintained in $O(\log(\omega \cdot |sch(\Sigma)|))$ space.

Lower Bound. Let us now proceed with the NL-hardness. We first introduce the so-called looping operator, which will allow us to establish a generic complexity tool for proving lower bounds for the Chase termination problem. Notice that this tool will be used, not only for simple linear TGDs, but also for all the other languages considered in this work. In fact, the goal of the looping operator is to provide a generic reduction from propositional atom entailment to the complement of Chase termination. Recall that an instance of propositional atom entailment consists of a database D, a set Σ of TGDs, and a

propositional (i.e., 0-ary) predicate q, and the question is whether $D \cup \Sigma \models q$, or, equivalently, whether q belongs to the result of the Chase of D w.r.t. Σ.

Let (D, Σ, q) be an instance of propositional atom entailment. Given an atom $A = p(\mathbf{t})$, where $\mathbf{t} = (t_1, \ldots, t_n)$, occurring either in D (i.e., $\mathbf{t} \in \mathbf{C}^n$) or in Σ (i.e., $\mathbf{t} \in \mathbf{V}^n$), we define, for some $Y \in \mathbf{V}$ not in Σ, the atomic formula

$$A_{\Sigma}^{Y} = \begin{cases} \exists X_{t_1} \ldots \exists X_{t_n}\, p(Y, X_{t_1}, \ldots, X_{t_n}), & \mathbf{t} \in \mathbf{C}^n, \\ p(Y, t_1, \ldots, t_n), & \mathbf{t} \in \mathbf{V}^n, \end{cases}$$

where $X_{t_1}, \ldots, X_{t_n} \in \mathbf{V}$ do not appear in Σ. Let $\Phi_{D,\Sigma}^{Y} = (\bigwedge_{A \in D} A_{\Sigma}^{Y})$, and Σ^Y be the set of TGDs obtained by replacing each atom A occurring in Σ with A_{Σ}^{Y}. We are now ready to define the looping operator.

Definition 4.35. *Consider an instance (D, Σ, q) of propositional atom entailment. The application of the* looping operator *on (D, Σ, q) returns the set of TGDs*

$$\begin{aligned} \mathsf{Loop}(D, \Sigma, q) \ &= \\ &\{loop(X, Y) \rightarrow \Phi_{D,\Sigma}^{Y}\} \cup \Sigma^{Y} \cup \{q(Y) \rightarrow \exists Z\, loop(Y, Z)\}, \end{aligned}$$

where $loop \notin sch(\Sigma)$. A class of TGDs Ł is closed under looping *if, for every instance (D, Σ, q) of propositional atom entailment, where $\Sigma \in \text{Ł}$, $\mathsf{Loop}(D, \Sigma, q) \in \text{Ł}$.* □

By using the looping operator, we can transfer, in a uniform way, lower bounds from propositional atom entailment to Chase termination. Our generic complexity result follows:

Proposition 4.36. *Let Ł be a class of TGDs that is closed under looping, such that propositional atom entailment for $(\mathsf{CT}^{\star} \cap \text{Ł})$, where $\star \in \{\mathsf{obl}, \mathsf{sobl}\}$, is $\mathcal{C}$-hard, for a complexity class $\mathcal{C}$ that is closed under log-space reductions. For a set $\Sigma \in \text{Ł}$, deciding whether $\Sigma \in \mathsf{CT}^{\star}$ is co$\mathcal{C}$-hard.* □

To establish the above generic result, it suffices to reduce propositional atom entailment under $(\mathsf{CT}^{\star} \cap \text{Ł})$ to the complement of Chase termination under Ł. In particular, given a (non-empty) database D, a set $\Sigma \in (\mathsf{CT}^{\star} \cap \text{Ł})$, and a propositional predicate q, we need to construct in log-space a set $\Sigma' \in \text{Ł}$ such that, $D \cup \Sigma \models q$ iff there exists a database D' such that a non-terminating

$\star$-chase sequence of D' w.r.t. Σ' exists. It can be shown that the above equivalence holds for $\Sigma' = \mathsf{Loop}(D, \Sigma, q)$. The key idea underlying the looping operator can be sketchily described as follows. Consider the simple linear TGD $r = loop(X,Y) \rightarrow \exists Z\, loop(Y,Z)$. It is easy to verify that there exists only one $\star$-chase sequence of $\{loop(a,b)\}$ w.r.t. $\{r\}$, which is non-terminating. Our intention is to mimic the behavior of r using Σ', with the key difference that an atom of the form $loop(t', t'')$ is obtained by applying r on an atom $loop(t,t')$ only if $D \cup \Sigma \models q$. This is achieved by "plugging" between $body(r)$ and $head(r)$ the set Σ^Y, which, by hypothesis, guarantees the termination of the Chase. The given database D is generated by the TGD $loop(X,Y) \rightarrow \Phi^Y_{D,\Sigma}$, while the check whether q is entailed is performed by $q(Y) \rightarrow \exists Z\, loop(Y,Z)$. Since, by assumption, Ł is closed under looping, $\Sigma' \in$ Ł, and Proposition 4.36 follows.

By the Immerman-Szelepcsényi theorem, coNL = NL. Thus, to obtain the NL-hardness for the Chase termination problem under simple linear TGDs, since SL is closed under looping, by Proposition 4.36, it suffices to show that propositional atom entailment under ($\mathsf{CT}^\star \cap \mathsf{SL}$) is NL-hard, even for unary and binary predicates. This is shown by giving a reduction from graph reachability. Given a directed graph $G = (N, E)$ and two nodes $s, t \in N$, we construct a database D, a set $\Sigma \in \mathsf{SL}$, and a propositional predicate q such that $D \cup \Sigma \models q$ iff t is reachable from s. The idea is to construct Σ in such a way that its predicate graph coincides with G, while D stores the node s, and q represents the node t. The next result follows:

Lemma 4.37. *Propositional atom entailment under* ($\mathsf{CT}^\star \cap \mathsf{SL}$)*, where* $\star \in \{\mathsf{obl}, \mathsf{sobl}\}$*, is* NL*-hard, even for unary and binary predicates.* □

Theorem 4.33 follows from Proposition 4.36, and Lemmas 4.34 and 4.37. It is interesting to say that $\mathsf{Loop}(D, \Sigma, q)$ belongs to ID and $\mathsf{DL\text{-}Lite}^{\mathsf{TGD}}$, and thus Theorem 4.33 holds also for inclusion dependencies and DL-Lite$_{\mathcal{R}}$.

Linear TGDs

We now focus on arbitrary linear TGDs, and we show the following:

Theorem 4.38. *Consider a set* $\Sigma \in \mathsf{L}$*. The problem of deciding whether* $\Sigma \in \mathsf{CT}^\star$*, where* $\star \in \{\mathsf{obl}, \mathsf{sobl}\}$*, is* PSPACE*-complete, and* NL*-complete for predicates of bounded arity.* □

Upper Bound. By Theorems 4.26 and 4.32, it suffices to show that the problem of deciding whether Σ is critically-richly-acyclic (or critically-weakly-

acyclic) can be solved in polynomial space, in general, and in nondeterministic logarithmic space, in case of predicates of bounded arity.

Lemma 4.39. *Consider a set* $\Sigma \in \mathsf{L}$. *The problem of deciding if* $\Sigma \in L$, *where* $L \in \{\mathsf{LCriticalRA}, \mathsf{LCriticalWA}\}$, *is in NSPACE*$(\omega \log(\omega \cdot |sch(\Sigma)|) + \omega \log(\omega \cdot |\Sigma|))$, *where* ω *is the maximum arity over all predicates of* $sch(\Sigma)$. □

The above technical lemma is shown by conceiving the complement of our problem as an extended version of graph reachability. In particular, we need to decide whether there exists a node v in the (extended) dependency graph of Σ that is reachable from itself via a critical cycle that contains a special edge. As for Lemma 4.34, this can be done via a nondeterministic procedure. However, in order to check for the criticality of the traversed cycle, apart from the two consecutive edges, the origin of the cycle, and the binary flag, we also need to remember the resolvent of the TGDs that label the visited edges. Such a resolvent can be computed and maintained in $O(\omega \log(\omega \cdot |sch(\Sigma)|) + \omega \log(\omega \cdot |\Sigma|))$ space, and its criticality can be checked using the same space.

Lower Bound. The NL-hardness is immediately inherited from Theorem 4.33. Concerning the PSPACE-hardness, since L is closed under looping, by Proposition 4.36, it suffices to show that propositional atom entailment under $(\mathsf{CT}^\star \cap \mathsf{L})$ is PSPACE-hard. This is shown by a reduction from the acceptance problem of a polynomial space Turing machine M.

Lemma 4.40. *Propositional atom entailment under* $(\mathsf{CT}^\star \cap \mathsf{L})$, *where* $\star \in \{\mathsf{obl}, \mathsf{sobl}\}$, *is* PSPACE-*hard.* □

Theorem 4.38 follows from Proposition 4.36, and Lemmas 4.39 and 4.40.

4.3.2 (Weak-)Guardedness

We proceed to investigate the (semi-)oblivious Chase termination problem for guarded and weakly-guarded TGDs. Although there is no way (at least an obvious one) to syntactically characterize the classes $(\mathsf{CT}^\star \cap \mathsf{WG})$ and $(\mathsf{CT}^\star \cap \mathsf{G})$, where $\star \in \{\mathsf{obl}, \mathsf{sobl}\}$, via rich- and weak-acyclicity, as we did for (simple) linear TGDs, it is possible to show that the problem of recognizing the above classes is decidable.

For technical reasons, we focus on *standard databases*, that is, databases that have at least two constants, let's say 0 and 1, that are available via the unary predicates $0(\cdot)$ and $1(\cdot)$, respectively. The results presented below, unless stated otherwise, hold only for standard databases. We show the following:

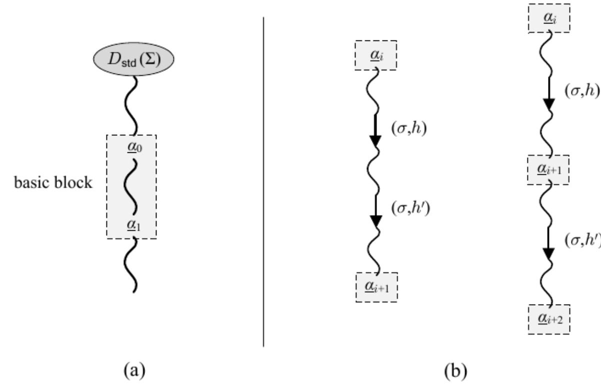


Fig. 4.4: Infinite Chase derivation

Theorem 4.41. *Consider a set $\Sigma \in \mathsf{WG}$. The problem of deciding whether $\Sigma \in \mathsf{CT}^\star$, where $\star \in \{\mathsf{obl}, \mathsf{sobl}\}$, is 2*EXPTIME*-complete, and* EXPTIME*-complete for predicates of bounded arity. The same holds even if $\Sigma \in \mathsf{G}$.* □

The goal of this section is to establish the above result.

Infinite Chase Derivations

We first focus our attention on weakly-guarded TGDs, and show that the Chase termination problem is decidable; this implies that the problem is decidable also for guarded TGDs.

By Proposition 4.13, given a set $\Sigma \in \mathsf{WG}$, to decide whether $\Sigma \notin \mathsf{CT}^\star$ (recall that we focus on standard databases), where $\star \in \{\mathsf{obl}, \mathsf{sobl}\}$, is tantamount to the problem of deciding whether Σ admits an infinite $\star$-chase derivation that starts from a standard database i.e., there exist sequences $I_0, I_1, \ldots$ and $(r_0, h_0), (r_1, h_1), \ldots$ as in Definition 4.12 with I_0 be a standard database. In what follows, we show that the latter is a decidable problem by presenting an alternating algorithm, called $\star$-InfiniteDerivation.

A General Description. One may be tempted to claim that a set of TGDs Σ admits an infinite $\star$-chase derivation that starts from a standard database iff Σ admits an infinite $\star$-chase derivation that starts from the critical database $D_{\mathsf{c}}(\Sigma)$. However, this is not true since $D_{\mathsf{c}}(\Sigma)$ is *not* a standard database. Nevertheless, the notion of the critical database can be naturally extended to the critical standard database, which is the standard database

consisting of all possible atoms that can be formed using predicates of $sch(\Sigma)$ and the constants 0 and 1. Formally, the *critical standard database* for a set Σ of TGDs (not necessarily weakly-guarded), denoted $D_{\mathsf{std}}(\Sigma)$, is defined as the database

$$\{0(0), 1(1)\} \ \cup \ \{p(\mathbf{t}) \mid p/n \in sch(\Sigma), \mathbf{t} \in \{0,1\}^n\}.$$

It is clear that the size of $D_{\mathsf{std}}(\Sigma)$ is exponential in general, and polynomial when the maximum arity over all predicates of $sch(\Sigma)$ is fixed. By giving a proof similar to the one in [63] for the critical database, we show the following:

Lemma 4.42. *Consider a set Σ of TGDs. It holds that, Σ admits an infinite $\star$-chase derivation that starts from a standard database, where $\star \in \{\mathsf{obl}, \mathsf{sobl}\}$, iff Σ admits an infinite $\star$-chase derivation that starts from $D_{\mathsf{std}}(\Sigma)$.* □

Our alternating algorithm, starting from an atom of $D_{\mathsf{std}}(\Sigma)$, and applying nondeterministically Chase steps, identifies a finite basic block of a Chase derivation (if it exists), which can then be repeated and give rise to an infinite Chase derivation; this is graphically illustrated in Figure 4.4(a). In other words, the algorithm tries to identify an atom A_0 from which, after applying some valid (depending on the version of the Chase) triggers, an atom A_1 isomorphic to A_0 is obtained — by isomorphism we mean that, starting from A_0 and A_1, we obtain isomorphic atoms. The segment of the derivation between A_0 and A_1 is the basic block that we can repeat infinitely many times, and obtain an infinite Chase derivation. Before giving the technical details about our algorithm, we need to briefly recall some auxiliary notions and results.

Auxiliary Notions and Results. It is well-known that a set $\Sigma \in \mathsf{WG}$ can be effectively transformed into a set $\Sigma' \in \mathsf{WG}$ such that all the TGDs of Σ' are single-head [22]. It is not difficult to verify that this transformation preserves Chase termination, i.e., $\Sigma \in \mathsf{CT}^{\star}$ iff $\Sigma' \in \mathsf{CT}^{\star}$, where $\star \in \{\mathsf{obl}, \mathsf{sobl}\}$. Henceforth, for technical clarity, we focus on TGDs with just one atom in the head. Let D be a database, and Σ a set of TGDs. Fix a $\star$-chase sequence $I_0 = D, I_1, \ldots$ of D w.r.t. Σ, for $\star \in \{\mathsf{obl}, \mathsf{sobl}\}$. The instance $\cup_{i \geqslant 0} I_i$, denoted $\star$-*chase*(D, Σ), can be naturally represented as a labeled directed graph $G = (N, E, \lambda)$ as follows: (1) for each atom $A \in \star\text{-}chase(D, \Sigma)$, there exists $v \in N$ such that $\lambda(v) = A$; (2) for each $i \geqslant 0$, with $I_i\langle r, h\rangle I_{i+1}$, and for each atom $A \in h(body(r))$, there exists $(v, u) \in E$ such that $\lambda(v) = A$ and $\{\lambda(u)\} = I_{i+1} \setminus I_i$; and (3) there are no other nodes and edges in G. The *guarded Chase forest* of D and Σ, denoted $\mathsf{gcf}(D, \Sigma)$, is the forest obtained from G by keeping only

the nodes associated with guard atoms, and their children; for more details, we refer the reader to [22].

Lemma 4.42 implies that our algorithm has to identify an infinite path in $\mathsf{gcf}(D_{\mathsf{std}}(\Sigma), \Sigma)$. This is achieved by constructing nondeterministically such a path, starting from an atom of $D_{\mathsf{std}}(\Sigma)$, until a basic block that can be repeated is identified. During this process, our algorithm exploits two key results established in [22], where the problem of query answering under (weakly-)guarded TGDs is investigated. Let us recall those results, and explain how they are applied; let D be an arbitrary database:

1. The subtree of $\mathsf{gcf}(D, \Sigma)$ rooted at some atom A is determined by the so-called cloud of A (modulo renaming of nulls) [22, Theorem 5.16]. The *cloud* of A w.r.t. D and Σ, denoted $cloud(A, D, \Sigma)$, is defined as

   $$\{B \mid B \in \star\text{-}chase(D, \Sigma) \text{ and } dom(B) \subseteq (dom(D) \cup dom(A))\},$$

   i.e., the atoms occurring in the result of the Chase with constants from D and terms from A. This result allows us to build the relevant path of $\mathsf{gcf}(D, \Sigma)$. In fact, an atom A on this path can be generated by considering only its parent atom A' and the cloud of A' w.r.t. D and Σ. Whenever a new atom is This result allows us to build the relevant path of $\mathsf{gcf}(D_{\mathsf{c}}(\Sigma), \Sigma)$. In fact, an atom A on this path can be generated by considering only its parent atom A' and the cloud of A' w.r.t. $D_{\mathsf{c}}(\Sigma)$ and Σ. Whenever a new atom is generated, we nondeterministically guess its cloud, and verify in a parallel universal computation of our algorithm that indeed belongs to the result of the Chase.
2. There exists a bound δ, which is double-exponential in the maximum arity ω of $sch(\Sigma)$ (and only ω appears in the second exponent), up to which we have to construct the relevant path of $\mathsf{gcf}(D_{\mathsf{c}}(\Sigma), \Sigma)$ in order to guarantee that all the obtained atoms are non-isomorphic. This implies that, for our purposes, we simply need to construct the path up to depth $(2 \cdot \delta)$. We use this fact to ensure that our algorithm terminates.

Let us clarify that in [22] only the oblivious Chase has been considered, and the above results have been explicitly established for the oblivious Chase. Nevertheless, it is not difficult to extend these results to the semi-oblivious Chase.

The Alternating Algorithm. We have now all the ingredients in place that are needed to define our algorithm. Given a set $\Sigma = \{r_1, \ldots, r_n\}$ as input, $\star$-$\mathsf{InfiniteDerivation}(\Sigma)$ consists of the following steps:

1. $Cl := D_{\mathsf{c}}(\Sigma)$, $H_{r_i} := \varnothing$, for each $i \in [n]$, $\mathit{flag} := 0$ and $\mathit{ctr} := 0$.
2. Guess an atom $A \in D_{\mathsf{std}}(\Sigma)$.
3. Guess a TGD $r \in \Sigma$, and a trigger (r, h) for Σ on Cl, where $h(\mathit{guard}(r)) = A$ and $h \diamond_r^\star h'$, for each $h' \in H_r$; if there is no such a trigger, then *reject*.
4. Let A be the atom obtained by applying (r, h) to Cl, and guess the cloud Cl of A w.r.t. $D_{\mathsf{std}}(\Sigma)$ and Σ.
5. Universally goto steps 6 and 7.
6. If Cl is a valid cloud, then *accept*; otherwise, *reject*.
7. $H_r := (H_r \cup \{h\}) \setminus H_{r,A}$, where $H_{r,A} \subseteq H_r$ is the set of homomorphisms that map at least one variable of $\mathit{var}(\mathit{body}(r))$ to a term not in $\mathit{dom}(A)$.
8. If $\mathit{flag} = 0$, then guess to apply or skip the following:
   a) $\mathit{loop} := (r, A, Cl)$ and $\mathit{flag} := 1$.
   b) $\mathit{nulls} := \mathit{invent}(A)$, where the latter is the set of nulls invented in A; if $\mathit{nulls} = \varnothing$, the *reject*.
   c) Goto step 10.
9. If $\mathit{flag} = 1$, then do the following:
   a) $\mathit{nulls} := (\mathit{dom}(A) \cap \mathit{nulls}) \cup \mathit{invent}(A)$.
   b) If $(\mathit{dom}(A) \cap \mathit{nulls}) = \varnothing$, then *reject*.
   c) If (r, A, Cl) and loop are the same (modulo bijective null renaming), then *accept*.
10. If $\mathit{ctr} = (2 \cdot \delta)$, then *reject*; otherwise, $\mathit{ctr} := \mathit{ctr} + 1$ and goto step 3.

By construction, $\star$-$\mathsf{InfiniteDerivation}(\Sigma)$, starting from an atom $A \in D_{\mathsf{std}}(\Sigma)$, identifies a basic block on a path P in the subtree of $\mathsf{gcf}(D_{\mathsf{std}}(\Sigma), \Sigma)$ rooted at A, which can be repeated infinitely many times. This allows us to safely conclude that P is an infinite path (or Chase derivation), and the algorithm accepts; if such a derivation does not exist, the algorithm terminates and rejects. It remains to explain why this derivation is a valid one, i.e., it does not contain conflicting triggers (depending on the version of the Chase).

Consider two triggers (r, h) and (r', h') occurring in the obtained infinite $\star$-chase derivation. There are two possible cases: either they occur in the same or in different basic blocks; this is illustrated in Figure 4.4(b). In the first case, $h \diamond_r^\star h'$ is guaranteed by construction; this is the reason why the set H_r is maintained during the execution of the algorithm, which stores all the "dangerous" homomorphisms that have been used to trigger r. In the second

case, $h \diamond_r^\star h'$ is guaranteed since the atom $h'(guard(r))$ necessarily contains a null that does not appear in the atom $h(guard(r))$; this is why the set *nulls* is maintained, which actually stores the nulls that can only appear in the atoms of a certain basic block. From the above discussion, we get the desired result:

Proposition 4.43. *Consider a set $\Sigma \in \mathsf{WG}$. It holds that, Σ admits an infinite $\star$-chase derivation that starts from a standard database, where $\star \in \{\mathsf{obl}, \mathsf{sobl}\}$, iff $\star$-$\mathsf{InfiniteDerivation}(\Sigma)$ accepts.* □

Complexity

Upper Bounds. By Propositions 4.13 and 4.43, we get that, for a set $\Sigma \in \mathsf{WG}$, $\Sigma \notin \mathsf{CT}^\star$ iff $\star$-$\mathsf{InfiniteDerivation}(\Sigma)$ accepts, where $\star \in \{\mathsf{obl}, \mathsf{sobl}\}$. Therefore, to establish the desired upper bounds, it suffices to show that our alternating algorithm runs in exponential space, in general, and in polynomial space, in the case of predicates of bounded arity; recall that AEXPSPACE = 2EXPTIME and APSPACE = EXPTIME. To this end, we show that the space required for the following tasks is exponential in the maximum arity ω of $sch(\Sigma)$, and polynomial in all the other parameters of the input: (1) maintain $D_{\mathsf{std}}(\Sigma)$ and the cloud of an atom; (2) maintain the set H_r, where $r \in \Sigma$; (3) maintain the integer value of *ctr*; and (4) verify that the guessed cloud is valid.

Lemma 4.44. *The algorithm $\star$-$\mathsf{InfiniteDerivation}$, where $\star \in \{\mathsf{obl}, \mathsf{sobl}\}$, runs in double-exponential time, in general, and in exponential time, for predicates of bounded arity.* □

The upper bounds of Theorem 4.41 follow from Propositions 4.13 and 4.43, and Lemma 4.44.

Lower Bounds. To establish the desired lower bounds, since G is closed under looping, by Proposition 4.36, it suffices to show the following:

Lemma 4.45. *Propositional atom entailment under $(\mathsf{CT}^\star \cap \mathsf{G})$, where $\star \in \{\mathsf{obl}, \mathsf{sobl}\}$, is 2EXPTIME-hard, and EXPTIME-hard for predicates of bounded arity.* □

The 2EXPTIME-hardness is obtained by a significant modification of the proof of Theorem 6.2 in [22], which shows the 2EXPTIME-hardness of propositional atom entailment under arbitrary guarded TGDs (not necessarily in $\mathsf{CT}^\star$). That proof simulates an AEXPSPACE Turing machine that uses no more

than 2^n worktape cells; this assumption can be made without affecting the generality of the proof. For proving Lemma 4.45, we make, w.l.o.g., an additional assumption: we assume the machine contains a counter of 2^{n-1} bits (i.e., the second half of the tape) that is initialized to zero and can count from 0 up to $(2^{2^{n-1}}-1)$. The counter is incremented by one until either the Turing machine stops, or it reaches the maximal value of $(2^{2^{n-1}}-1)$, in which case the machine is forced to stop in a rejecting state. This makes sure that the machine cannot cycle and always stops within $O(2^{2^n})$ steps. Adding counters to Turing machines, giving rise to the concept of *clocked Turing machines*, is a well-known technique; see [59, 74]. The fact that we consider a clocked Turing machine, together with the fact that we focus on standard databases, allows us to construct the double-exponentially many configurations of the machine using a set of TGDs that ensures the termination of the Chase, which is not the case in the proof of Theorem 6.2 of [22]. By following a similar approach, we can also show the EXPTIME-hardness in the case of predicates of bounded arity, and Lemma 4.45 follows.

Non-Standard Databases

From the above discussion, it is clear that standard databases are crucial for establishing the lower bounds in Proposition 4.45; in particular, to guarantee that the sets of guarded TGDs employed in the reductions are indeed members of $(\mathsf{CT}^\star \cap \mathsf{G})$, where $\star \in \{\mathsf{obl}, \mathsf{sobl}\}$. Interestingly, the upper bounds stated in Theorem 4.41 hold also for non-standard databases. This can be shown by slightly modifying $\star$-InfiniteDerivation in such a way that, instead of starting from $D_{\mathsf{std}}(\Sigma)$, where $\Sigma \in \mathsf{WG}$ is the given set of TGDs, starts from the critical database $D_{\mathsf{c}}(\Sigma)$. After applying this modification, it is easy to see that Σ admits an infinite $\star$-chase derivation (that starts from an arbitrary, not necessarily standard database) iff $\star$-InfiniteDerivation(Σ) accepts, and we immediately get the following result for arbitrary databases:

Theorem 4.46. *Consider a set* $\Sigma \in \mathsf{WG}$*. The problem of deciding whether* $\Sigma \in \mathsf{CT}^\star$*, where* $\star \in \{\mathsf{obl}, \mathsf{sobl}\}$*, is in 2*EXPTIME*, and in* EXPTIME *for predicates of bounded arity.* □

The exact complexity of the Chase termination problem in case of arbitrary (not necessarily standard databases) is still open.

4.4 Dealing with EGDs

Most of the work in the literature has focused on the problem of checking if all Chase sequences (for some variation of the Chase) are terminating, independently of the considered database. However, since in many applications the ultimate goal is to compute a universal model, checking for the *existence* of a terminating Chase sequence and constructing it suffices for the purpose.

In this regard, a universal model might be computed using the *core Chase* [33], which is a variant of the standard Chase where all applicable Chase steps are fired "in parallel", rather than picking one non-deterministically as in the standard Chase. One consequence of the parallel application is that nondeterminism is eliminated. Another important property of the core Chase is that it is complete for finding universal models, that is, whenever a universal model exists, the core Chase terminates and finds such a model. Thus, if we know that there exists a terminating standard Chase sequence (and thus a universal model), then we can use the core Chase to compute a universal model.

Furthermore, the weaker requirement of checking for the existence of a terminating Chase sequence, rather than ensuring that every Chase sequence is terminating, can be profitably leveraged to identify more sets of dependencies for which we can compute a universal model. For instance, the set of dependencies $\Sigma_{1.2}$ of Example 1.2 might be identified by a criterion ensuring termination of at least one Chase sequence. However, every criterion requiring all Chase sequences to be terminating will not recognize $\Sigma_{1.2}$, thereby providing no information about whether we can compute a universal model.

Despite the significant body of work in this area, there are still large classes of dependencies for which the Chase is not applicable as termination cannot be statically established.

One weakness of current approaches is that the analysis of EGDs is limited or absent altogether. In fact, more general approaches, such as *super-weak acyclicity* [63], semi-dynamic approaches [50], and rewriting approaches [54, 55, 56], were meant to guarantee termination of TGDs only. Other approaches, such as *weak acyclicity* [38] and *safety* [66], guarantee the termination of a set of TGDs and EGDs, but do not analyze EGDs at all, which leads them to impose strong conditions on TGDs to guarantee termination. Firing relations among dependences used in stratification-based approaches [33, 66, 55] consider EGDs in a limited way. To mitigate the aforementioned issues, an "indirect" way of dealing with EGDs was proposed

in [48, 63], where a set Σ of TGDs and EGDs is rewritten into a set Σ' containing only TGDs, and termination analysis is carried out on Σ'. The aim is to "simulate" the behavior of the EGDs by means of TGDs. While these preprocessing steps ensure soundness, i.e., if all Chase sequences of Σ' are terminating then all Chase sequences of Σ are terminating, they are not complete, i.e., the implication in the opposite direction does not hold. The first approach of this kind, known as *natural simulation*, has been proposed in [48], and further refined by the *substitution-free simulation* in [63].

Treating EGDs as first-class citizens is very important, as they are among the most popular classes of dependencies in real applications, playing a critical role in maintaining data integrity, query optimization and indexing, and schema design [37]. For instance, functional dependencies can be expressed by EGDs. In very simple scenarios, such as Example 1.2 above, current termination criteria are not able to say whether a universal model can be found. As a further scenario, Example 4.48 shows a simple set of dependencies for which all Chase sequences are terminating, but there is no terminating Chase sequence for the set of dependencies obtained from the EGD simulation.

In this section, we propose new sufficient conditions ensuring that a set of dependencies (possibly containing both TGDs and EGDs) admits at least one terminating Chase sequence, independently of the database. This approach performs an explicit analysis of EGDs and identifies sets of dependencies that are not captured by any of the current techniques. To the best of our knowledge, sufficient conditions ensuring termination of at least one Chase sequence was studied only in [65, 66].

We start by shedding light on the relationships between the classes $\mathsf{CT}^{\star}_{\mathsf{q}}$, where $\mathsf{q} \in \{\forall, \exists\}$ and $\star \in \{\mathsf{obl}, \mathsf{sobl}, \mathsf{std}, \mathsf{core}\}$, when arbitrary sets of TGDs and EGDs are considered. Recall that a hierarchy for sets consisting only of TGDs has been presented in this chapter, but the relationships in the presence of both TGDs and EGDs have not been studied so far. We also discuss different issues arising in the presence of EGDs.

Given two sets C_1 and C_2, we write $C_1 \nparallel C_2$ iff $C_1 \not\subseteq C_2$ and $C_2 \not\subseteq C_1$.

Theorem 4.47. *For general dependencies (including TGDs and EGDs), the following relations hold:*

1. $\mathsf{CT}^{\star}_{\forall} \subsetneq \mathsf{CT}^{\star}_{\exists}$ *for* $\star \in \{\mathsf{obl}, \mathsf{sobl}, \mathsf{std}\}$*, and* $\mathsf{CT}^{\mathsf{core}}_{\forall} = \mathsf{CT}^{\mathsf{core}}_{\exists}$*;*
2. $\mathsf{CT}^{\mathsf{obl}}_{\mathsf{q}} \subsetneq \mathsf{CT}^{\mathsf{sobl}}_{\mathsf{q}} \subsetneq \mathsf{CT}^{\mathsf{std}}_{\mathsf{q}} \subsetneq \mathsf{CT}^{\mathsf{core}}_{\mathsf{q}}$ *for* $\mathsf{q} \in \{\forall, \exists\}$*;*
3. $\mathsf{CT}^{\mathsf{obl}}_{\exists} \nparallel \mathsf{CT}^{\mathsf{sobl}}_{\forall}$*,* $\mathsf{CT}^{\mathsf{sobl}}_{\exists} \nparallel \mathsf{CT}^{\mathsf{std}}_{\forall}$*, and* $\mathsf{CT}^{\mathsf{obl}}_{\exists} \nparallel \mathsf{CT}^{\mathsf{std}}_{\forall}$. □

TGDs	TGDs and EGDs
$\mathsf{CT}^{\mathsf{obl}}_{\forall} = \mathsf{CT}^{\mathsf{obl}}_{\exists}$ $\mathsf{CT}^{\mathsf{sobl}}_{\forall} = \mathsf{CT}^{\mathsf{sobl}}_{\exists}$	$\mathsf{CT}^{\mathsf{obl}}_{\forall} \subsetneq \mathsf{CT}^{\mathsf{obl}}_{\exists}$ $\mathsf{CT}^{\mathsf{sobl}}_{\forall} \subsetneq \mathsf{CT}^{\mathsf{sobl}}_{\exists}$
$\mathsf{CT}^{\mathsf{obl}}_{\exists} \subsetneq \mathsf{CT}^{\mathsf{sobl}}_{\forall}$ $\mathsf{CT}^{\mathsf{sobl}}_{\exists} \subsetneq \mathsf{CT}^{\mathsf{std}}_{\forall}$	$\mathsf{CT}^{\mathsf{obl}}_{\exists} \nparallel \mathsf{CT}^{\mathsf{sobl}}_{\forall}$ $\mathsf{CT}^{\mathsf{sobl}}_{\exists} \nparallel \mathsf{CT}^{\mathsf{std}}_{\forall}$ $\mathsf{CT}^{\mathsf{obl}}_{\exists} \nparallel \mathsf{CT}^{\mathsf{std}}_{\forall}$
$\mathsf{CT}^{\mathsf{std}}_{\forall} \subsetneq \mathsf{CT}^{\mathsf{std}}_{\exists}$	$\mathsf{CT}^{\mathsf{std}}_{\forall} \subsetneq \mathsf{CT}^{\mathsf{std}}_{\exists}$
$\mathsf{CT}^{\mathsf{core}}_{\forall} = \mathsf{CT}^{\mathsf{core}}_{\exists}$	$\mathsf{CT}^{\mathsf{core}}_{\forall} = \mathsf{CT}^{\mathsf{core}}_{\exists}$

Table 4.1: Relationships among the $\mathsf{CT}^{\star}_{\mathsf{q}}$'s classes.

The relationships between the classes $\mathsf{CT}^{\star}_{\mathsf{q}}$, where $\mathsf{q} \in \{\forall, \exists\}$ and $\star \in \{\mathsf{std}, \mathsf{obl}, \mathsf{sobl}, \mathsf{core}\}$, are shown in Table 4.1, for the case of TGDs only and in the presence of both TGDs and EGDs.

As discussed above, Chase termination criteria proposed in the literature focus on TGDs considering EGDs in a very limited way. More general approaches (including *SwA*, *LS*, *MFA*, *MSA*) as well as rewriting techniques were meant to guarantee termination of TGDs only.

The use of the substitution-free simulation allows to deal with EGDs in an indirect way, but as we are going to show, this is not enough to properly analyse the role of EGDs during the execution of the Chase. Below is an example showing how the substitution-free simulation works.

Example 4.48. Consider the following set of dependencies $\Sigma_{4.48}$ (containing both TGDs and EGDs):

$$
\begin{aligned}
r_1 &: \ a(X), b(X) \rightarrow c(X)\\
r_2 &: \ c(X) \rightarrow \exists Y\, a(X) \wedge b(Y)\\
r_3 &: \ c(X) \rightarrow \exists Y\, a(Y) \wedge b(X)\\
r_4 &: \ a(X), a(Y) \rightarrow X = Y\\
r_5 &: \ b(X), b(Y) \rightarrow X = Y
\end{aligned}
$$

The substitution-free simulation works as follows:

1. The TGDs below (equality-axioms) are added to $\Sigma_{4.48}$:

$$
\begin{aligned}
a_1 &: \ eq(X,Y) \rightarrow eq(Y,X)\\
a_2 &: \ eq(X,Y), eq(Y,Z) \rightarrow eq(X,Z)\\
a_{3.1} &: a(X) \rightarrow eq(X,X)\\
a_{3.2} &: b(X) \rightarrow eq(X,X)\\
a_{3.3} &: c(X) \rightarrow eq(X,X)
\end{aligned}
$$

2. Every occurrence of $X = Y$ in $\Sigma_{4.48}$ is replaced with $eq(X, Y)$. In our case, this affects r_4 and r_5 only, which are replaced with:

$$r'_4 : a(X), a(Y) \rightarrow eq(X, Y)$$
$$r'_5 : b(X), b(Y) \rightarrow eq(X, Y)$$

3. Dependency r_1, which contains multiple occurrences of X in the body, is (non-deterministically) replaced with one of the following two dependencies, where one of the two occurrences of X is replaced with X_2, and the atom $eq(X, X_2)$ is added to the body:

$$r'_1 : a(X_2), b(X), eq(X, X_2) \rightarrow c(X)$$
$$r''_1 : a(X), b(X_2), eq(X, X_2) \rightarrow c(X)$$

Notice that the only dependencies that remain unchanged are r_2 and r_3. Also, notice that there are no EGDs anymore in the resulting set of dependencies (their role is "simulated" by the rewriting). □

Although not explicitly stated, but somehow left implicit in [48, 63], the natural simulation and the substitution-free simulation ensure the desirable *soundness* property: if, for every database D, all $\star$-chase sequences of D with Σ' are terminating, then for every database D, all $\star$-chase sequences of D with Σ are terminating, for $\star \in \{\mathsf{obl}, \mathsf{sobl}, \mathsf{std}\}$. The natural question now is whether these simulations are also *complete*, that is, if the implication in the opposite direction holds. The answer is negative for both approaches, as stated in the following theorem. Furthermore, we show that the same properties hold when checking for the existence of at least one terminating $\star$-chase sequence. We focus on the substitution-free simulation only, as it is a refinement of the natural simulation.

Theorem 4.49. *Let Σ be a set of TGDs and EGDs and Σ' be a set of TGDs obtained from Σ by applying the substitution-free simulation. For every $\star \in \{\mathsf{obl}, \mathsf{sobl}, \mathsf{std}\}$ and every $\mathsf{q} \in \{\forall, \exists\}$,*

1. if $\Sigma' \in \mathsf{CT}^\star_\mathsf{q}$ then $\Sigma \in \mathsf{CT}^\star_\mathsf{q}$.
2. $\Sigma \in \mathsf{CT}^\star_\mathsf{q}$ does not imply $\Sigma' \in \mathsf{CT}^\star_\mathsf{q}$. □

The theorem above says that there are sets Σ of TGDs and EGDs such that $\Sigma \in \mathsf{CT}^\star_\mathsf{q}$ but their substitution-free simulation Σ' does not belong to $\mathsf{CT}^\star_\mathsf{q}$, and thus it is not possible to realize that $\Sigma \in \mathsf{CT}^\star_\mathsf{q}$ with an analysis of Σ'. The set of dependencies $\Sigma_{4.48}$ of Example 4.48 above is one of such

cases: $\Sigma_{4.48}$ belongs to $\mathsf{CT}^{\star}_{\forall}$ (and thus belongs to $\mathsf{CT}^{\star}_{\exists}$ too), but any of its substitution-free simulations is not even in $\mathsf{CT}^{\star}_{\exists}$, for every $\star \in \{\mathsf{obl}, \mathsf{sobl}, \mathsf{std}\}$. The problem is that the simulation of EGDs by means of TGDs is not able to fully capture the specific behavior of EGDs, which replace null values (with constants and other null values). This aspect is not faithfully modeled by storing the information that a null value is equal to a constant or to another null value.

In Sections 4.4.1 and 4.4.2, we propose approaches that perform a direct analysis of EGDs. However, dealing with EGDs needs some care. In some cases the presence of EGDs allows us to have a terminating $\star$-chase sequence when the set consisting only of the TGDs does not have one; at the same time, the opposite case can occur, that is, in the presence of EGDs there is no terminating $\star$-chase sequence while the set consisting only of the TGDs does have one, where $\star$ can be one of $\{\mathsf{obl}, \mathsf{sobl}, \mathsf{std}\}$. The following two examples show such cases.

Example 4.50. Consider the set of dependencies $\Sigma_{1.2}$ of Example 1.2 and the database $D = \{n(a)\}$. There is no terminating $\star$-chase sequence of $D_{1.2}$ with the set of TGDs $\Sigma'_{1.2} = \{r_1, r_2\}$, for every $\star \in \{\mathsf{obl}, \mathsf{sobl}, \mathsf{std}\}$. In fact, it is easy to see that an infinite number of facts is introduced: $e(a, z_1), n(z_1), e(z_1, z_2), \ldots$. However, the addition of the EGD r_3 allows us to have a terminating $\star$-chase sequence, obtained by enforcing first r_1 and then r_3, and whose result is the universal model $\{n(a), e(a, a)\}$. □

Example 4.51. Consider the set of dependencies $\Sigma_{4.51}$ below:

$$\begin{aligned} r_1 &: \ n(X) \rightarrow \exists Y\ \exists Z\ e(X, Y, Z) \\ r_2 &: \ e(X, Y, Y) \rightarrow n(Y) \\ r_3 &: \ e(X, Y, Z) \rightarrow Y = Z \end{aligned}$$

For every database D, every $\star$-chase sequence of D with the set of TGDs $\Sigma'_{4.51} = \{r_1, r_2\}$ is terminating, for every $\star \in \{\mathsf{obl}, \mathsf{sobl}, \mathsf{std}\}$. On the other hand, there is no terminating $\star$-chase sequence of $D = \{n(a)\}$ with $\Sigma_{4.51}$, as an infinite number of facts is introduced: $e(a, z_1, z_1)$, $n(z_1), e(z_1, z_2, z_2)$, $n(z_2), \ldots$. □

In the following, given a set of dependencies Σ, we use Σ_{tgd} and Σ_{egd} to denote the sets of all TGDs and all EGDs in Σ, respectively (obviously, $\Sigma = \Sigma_{tgd} \cup \Sigma_{egd}$). Furthermore, we use $\Sigma_{\forall}$ and $\Sigma_{\exists}$ to denote the set of all full dependencies in Σ (these include full TGDs and all EGDs) and the

set of all existentially quantified dependencies in Σ, respectively (obviously, $\Sigma = \Sigma_\forall \cup \Sigma_\exists$).

For a class $\mathcal{C}$ of sets of TGDs defined by some criterion (e.g., $\mathcal{S}w\mathcal{A}$ and $\mathcal{LS}$), we assume to also extend such class to TGDs and EGDs by applying the substitution-free simulation to the given set Σ of TGDs and EGDs, obtaining Σ' and then verifying whether $\Sigma' \in \mathcal{C}$.

4.4.1 Semi-Stratification

In this section, we introduce a new sufficient condition for checking if a set of dependencies belongs to $\mathsf{CT}^{\mathsf{std}}_{\exists}$. Our condition strictly generalizes stratification.

First of all, we recall the notion of stratification proposed in [33]. Given two dependencies r_1 and r_2, we write $r_1 \prec r_2$ iff there exist an instance K, an instance J, a homomorphism h_1 from $body(r_1)$ to K, and a homomorphism h_2 from $body(r_2)$ to J, such that:

- $K \models h_2(r_2)$,
- $K \xrightarrow{r_1,h_1,\gamma_1} J$ is a standard Chase step (for some γ_1), and
- $J \not\models h_2(r_2)$.

The *Chase graph* $G(\Sigma)$ of a set of dependencies Σ is a directed graph (Σ, E) containing an edge (r_1, r_2) iff $r_1 \prec r_2$. Then, Σ is *stratified* (*Str*) iff every cycle of $G(\Sigma)$ is weakly acyclic.

We now introduce a new relation between dependencies along with the corresponding graph it induces—they are used to define our criterion, allowing us to extend stratification.

Definition 4.52 (Activation graph). *Let Σ be a set of dependencies. Given two dependencies $r_1, r_2 \in \Sigma$, we write $r_1 < r_2$ iff there exist instances K and J, a homomorphism h_1 from body(r_1) to K, and a homomorphism h_2 from body(r_2) to J, such that:*

- $K \models h_2(r_2)$,
- $K \xrightarrow{r_1,h_1,\gamma_1} J$ *is a standard Chase step (for some γ_1),*
- $J \not\models h_2(r_2)$*, and*
- *if $r_2 \in \Sigma_\exists$, then $\nexists r_3 \in \Sigma_\forall$ such that $K \xrightarrow{r_3,h_3,\gamma_3} J'$ and $J' \models h_2(r_2)$ (for some h_3, γ_3).*

The activation graph *$G_f(\Sigma)$ of Σ is a directed graph (Σ, E_f) containing a directed edge (r_1, r_2) iff $r_1 < r_2$.*

We say that a dependency $r_1 \in \Sigma$ is fireable with respect to *Σ if there exists a dependency $r_2 \in \Sigma$ such that $r_2 < r_1$.* □

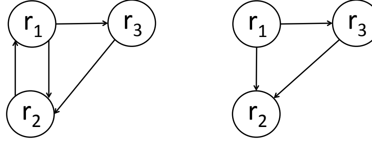


Fig. 4.5: Chase graph (left) and firing graph (right) of $\Sigma_{4.54}$.

Definition 4.53 (Semi-stratified dependencies). *A set of dependencies Σ is* semi-stratified *(*S-Str*) iff every strongly connected component of $G_f(\Sigma)$ is weakly acyclic.* □

Example 4.54. Consider the following set of TGDs $\Sigma_{4.54}$:

$$r_1: \ n(X) \rightarrow \exists Y \ e(X,Y)$$
$$r_2: \ e(X,Y) \rightarrow n(Y)$$
$$r_3: \ e(X,Y) \rightarrow e(Y,X)$$

The Chase and the firing graphs are depicted in Figure 4.5. Notice that, since r_2 and r_3 are full TGDs, their incoming edges are the same in the two graphs. On the other hand, the edge in $G(\Sigma_{4.54})$ from r_2 to r_1 does not belong to $G_f(\Sigma_{4.54})$, as the firing of r_1 because of r_2 is blocked by first enforcing r_3. It can be easily verified that $\Sigma_{4.54}$ is semi-stratified, but not stratified.

Consider now the database $D = \{n(a)\}$. The standard Chase sequence consisting of the iterative application of r_1 followed by r_2 is non-terminating. However, if we apply r_3 before r_1, we obtain a terminating standard Chase sequence producing the instance $K = \{n(a), e(a, z_1), \ n(z_1), \ e(z_1, a)\}$. Such a standard Chase sequence is terminating as no more standard Chase steps can be added. □

Theorem 4.55. *For every semi-stratified set of dependencies Σ and for every database D, there exists a terminating standard Chase sequence of D with Σ whose length is polynomial in the size of D.* □

As the following theorem states, it can be decided in co-*NP* whether a set of dependencies is semi-stratified.

Theorem 4.56. *Deciding if a set of dependencies is semi-stratified is in* co-*NP.* □

The following theorem shows the relative expressivity of $\mathcal{S}$-$\mathcal{S}tr$ and other classes of dependencies previously proposed.

Theorem 4.57.

1. $\mathcal{S}tr \subsetneq \mathcal{S}\text{-}\mathcal{S}tr$.
2. $\mathcal{S}\text{-}\mathcal{S}tr \nparallel \mathcal{C}$ *for* $\mathcal{C} \in \{\mathcal{SC}, \mathcal{AC}, \mathcal{MFA}\}$. □

Notice that SC, AC, and MFA guarantee that all standard Chase sequences are terminating, while Str and $S\text{-}Str$ guarantee the existence of at least one terminating standard Chase sequence.

We recall that $\mathcal{SC} \subsetneq \mathcal{AC}$, and thus the incomparability of $\mathcal{S}\text{-}\mathcal{S}tr$ with $\mathcal{SC}$ and $\mathcal{AC}$ implies that $\mathcal{S}\text{-}\mathcal{S}tr$ is incomparable also with any other class included by $\mathcal{AC}$ and containing $\mathcal{SC}$ (e.g., $\mathcal{S}w\mathcal{A}$, $\mathcal{SR}$, and $\mathcal{IR}$)—see [56] for a complete picture.

4.4.2 Adornment Algorithm

In this section, we propose another decidable sufficient condition for a set of dependencies to be in $\mathsf{CT}_{\exists}^{\mathsf{std}}$.

Specifically, we propose an algorithm which takes as input a set of dependencies, and gives as output a set of adorned dependencies and a boolean value. The aim of the algorithm is twofold: *(i)* it defines a termination criterion on its own—on the basis of the boolean value returned by the algorithm; and *(ii)* it can be combined with other termination criteria to enhance them, in that (strictly) more sets of dependencies in $\mathsf{CT}_{\exists}^{\mathsf{std}}$ can be identified by using our algorithm in conjunction with a termination criterion—this is achieved by analyzing the set of adorned dependencies returned by the algorithm. Before presenting our approach, we introduce additional terminology and notation.

Adornments. An *adornment symbol* is an element of the alphabet $\Lambda = \{b\} \cup \{f_i \mid i \in \mathbb{N}\}$, where b is called "bound" symbol and the f_i's are called "free" symbols. Consider an n-ary predicate p. An *adornment* of p is a string α of length n built from adornment symbols; we call p^α an *adorned predicate*. An *adorned atom* is of the form $p^\alpha(\mathbf{t})$, where $p(\mathbf{t})$ is an atom and α is an adornment of p. An *adorned conjunction* is a conjunction of adorned atoms. An *adorned dependency* is a dependency containing adorned atoms. Given an adorned formula (i.e., atom, conjunction of atoms, dependency, etc.) or set of adorned formulas F, we use $src(F)$ to denote the formula or set of formulas derived from F by deleting all adornments. We also say that F is an *adorned version* of $src(F)$.

Given a set of adorned predicates AP, the set of the *adorned versions* of an atom $p(\mathbf{t})$ *w.r.t.* AP is defined as follows:

$$\mathcal{A}(p(\mathbf{t}), AP) = \{p^{\alpha}(\mathbf{t}) \mid p^{\alpha} \in AP\}$$

The set of the *adorned versions* of a conjunction of atoms $conj = A_1 \wedge \cdots \wedge A_k$ *w.r.t.* AP is defined as follows:

$$\mathcal{A}(conj, AP) = \{A_1^{\alpha_1} \wedge \cdots \wedge A_k^{\alpha_k} \mid A_i^{\alpha_i} \in \mathcal{A}(A_i, AP) \text{ for } 1 \leq i \leq k\}$$

If $conj$ is the empty conjunction, then $\mathcal{A}(conj, AP)$ contains only the empty conjunction.

Given an adorned atom $R^{\alpha_1 \cdots \alpha_n}(t_1, ..., t_n)$, we say that t_i *is adorned with* α_i. An adorned atom or conjunction is *coherent* if every variable occurring in it is always adorned with the same adornment symbol and constants are adorned with b. For instance, the adorned conjunction $n^b(X), e^{f_1 b}(X, Y)$ is not coherent because X is adorned with b in the first atom and with f_1 in the second atom. On the other hand, $n^{f_1}(X), e^{f_1 b}(X, Y)$ is coherent.

An *adornment definition* is an expression of the form $f_i = f^r_Z(\alpha)$ where f_i is an adornment symbol, r is a TGD of the form $\varphi(\mathbf{X}, \mathbf{Y}) \rightarrow \exists \mathbf{Z}\, \psi(\mathbf{X}, \mathbf{Z})$, Z is in $\mathbf{Z}$, and α is a string of n adornment symbols with n being the number of variables in $\mathbf{X}$. The role of adornment definitions will be explained shortly.

Head adornment. One important step of our adornment algorithm is the propagation of adornments from the body to the head of dependencies, which is defined as follows. Given a set AD of adornment definitions, a dependency $r : body \rightarrow head$, and a coherent adorned version $body^{\mu}$ of $body$, we define $HeadAdn(r, body^{\mu}, AD)$ as the procedure that updates AD and returns an adorned version $head^{\mu}$ of $head()$ as follows:

1. if r is an EGD, then $head^{\mu} = head$, and AD is not modified.
2. Otherwise, r is a TGD $\varphi(\mathbf{X}, \mathbf{Y}) \rightarrow \exists \mathbf{Z}\, \psi(\mathbf{X}, \mathbf{Z})$ and $head^{\mu}$ is obtained from $\exists \mathbf{Z}\, \psi(\mathbf{X}, \mathbf{Z})$ as follows:
   - every universally quantified variable (i.e., every $X \in \mathbf{X}$) is adorned with the same adornment symbol the variable is adorned with in $body^{\mu}$ (notice that such an adornment symbol is unique as $body^{\mu}$ is coherent);
   - constants are adorned with b;
   - every (existentially quantified) variable $z \in \mathbf{Z}$ is adorned as follows.[1] Let $f^r_Z(\alpha)$ be the Skolem term where if $\mathbf{X} = X_1, ..., X_n$ then $\alpha =$

[1] It is assumed that the existentially quantified variables are considered one at a time following the order they appear in $\mathbf{Z}$. Also, an arbitrary but fixed ordering of the variables in $\mathbf{X}$ is assumed.

$\alpha_1, ..., \alpha_n$ is the string of adornment symbols such that every X_j is adorned with α_j in $body^\mu$, for $1 \leq j \leq n$. If an adornment definition of the form $f_i = f_Z^r(\alpha)$ is already in AD, then Z is adorned with f_i and AD is not modified. Otherwise, Z is adorned with f_j, where $j = 1 + \max\{k \mid f_k \text{ appears in } AD\}$, and $f_j = f_Z^r(\alpha)$ is added to AD.

For instance, assuming $AD = \varnothing$ and given a TGD $r: \;\; r(X,Y) \rightarrow \exists Z \; r(X,Z)$ we have that $HeadAdn(r, r^{bb}(X,Y), AD)$ gives the adorned formula $\exists Z \, r^{bf_1}(X,Z)$ and $f_1 = f_Z^r(b)$ is added to AD.

Cyclic adornment symbol. Given a set of adornment definitions AD, we use $\Omega(AD)$ to denote the labeled directed graph whose vertices are the adornment symbols appearing in AD, and where there is a directed edge from f_i to f_j labeled with f_Z^r iff there are $f_i = f_Z^r(\cdots f_j \cdots)$ and $f_j = f_W^s(\cdots)$ in AD with $r, s \in \Sigma_\exists$ and there are $r_1, ..., r_n \in \Sigma_\forall$ $(n \geq 0)$ such that $s < r_1 < \cdots < r_n < r$.

An adornment symbol f_i is *cyclic w.r.t. AD* if there is a path in $\Omega(AD)$ departing from f_i where (at least) two edges have the same label. We say that an adorned head $\exists \mathbf{Z} \, \psi^\mu(\mathbf{X}, \mathbf{Z})$ is *cyclic* (*w.r.t. AD*) if there is a variable z in $\mathbf{Z}$ adorned with a cyclic adornment symbol.

Adornment Substitution. An *adornment substitution* θ is a set of pairs of the form f_i/f_j (whose intuitive meaning is that f_i is replaced by f_j), where f_i and f_j are adornment symbols such that if $f_i/f_j \in \theta$ then there is no f_j/f_k in θ (that is, a symbol f_j used to replace a symbol f_i cannot be substituted by a symbol f_k). The result of applying θ to an adornment α, denoted $\alpha\theta$, is the adornment obtained from α by simultaneously replacing every occurrence in α of an adornment symbol f_i with f_j iff $f_i/f_j \in \theta$. This is extended to adorned atoms, adorned dependencies, adornment definitions, etc., in the obvious way.

Given a set of adornment definitions AD, an adornment substitution θ is *valid* (*w.r.t. AD*) if for every f_i/f_j in θ, it is the case that AD contains adornment definitions of the form $f_i = f_Z^r(\alpha)$ and $f_j = f_Z^r(\alpha')$.

Given a set of adorned dependencies Σ^μ and a dependency r, we define:

$$AP(\Sigma^\mu) = \{p^\alpha \mid p^\alpha(\mathbf{t}) \text{ appears in } \Sigma^\mu\}$$
$$D^\mu(\Sigma^\mu) = \{p(\alpha_1, ..., \alpha_n) \mid p^{\alpha_1 \cdots \alpha_n} \in AP(\Sigma^\mu)\}$$
$$B^\mu(r, \Sigma^\mu) = \{body^\mu \mid r^\mu \colon body^\mu \rightarrow head^\mu \in \Sigma^\mu \wedge src(r^\mu) = r\}$$

We are now ready to introduce the $Adn^\exists$ algorithm (Algorithm $Adn^\exists$). The input is a set of dependencies Σ, while the output is a set of adorned

Input: Set of dependencies Σ over schema $\mathcal{R}$.
Output: Set of adorned dependencies Σ^{μ}, Boolean value $Acyc$.

1: $Acyc = true$;
2: $\Sigma^{\mu} = \{p(X_1, ..., X_n) \rightarrow p^{b...b}(X_1, ..., X_n) \mid p \in \mathcal{R}$ and $arity(p) = n\}$;
3: $AD = \varnothing$;
4: **repeat**
5: $\quad \Sigma^{\mu}_{old} = \Sigma^{\mu}$;
6: $\quad$ **if** $\exists r \in \Sigma_{\forall}$ s.t. $\langle b, r^{\mu} \rangle = adorn(r)$ and $b = true$ **then**
7: $\quad\quad \Sigma^{\mu} = \Sigma^{\mu} \cup \{r^{\mu}\}$;
8: $\quad\quad$ **if** $r \in \Sigma_{egd}$ s.t. $D^{\mu}(\Sigma^{\mu}) \not\models r$ **then**
9: $\quad\quad\quad \tau = \{f_i/s\} = ChaseStep(r, D^{\mu}(\Sigma^{\mu}))$;
10: $\quad\quad\quad \Sigma^{\mu} = \Sigma^{\mu}\tau;\ AD = AD \setminus \{f_i = f^s_Z(\alpha) \in AD\};\ AD = AD\tau$;
11: $\quad$ **else if** $\exists r \in \Sigma_{\exists}$ s.t. $\langle b, r^{\mu} \rangle = adorn(r)$ and $b = true$ **then**
12: $\quad\quad \Sigma^{\mu} = \Sigma^{\mu} \cup \{r^{\mu}\}$;
13: $\quad$ **if** $\exists\, r^{\upsilon} \in \Sigma^{\mu} \wedge \exists$valid subst. $\theta \neq \varnothing$ s.t. $r^{\mu}\theta = r^{\upsilon} \wedge src(r^{\upsilon}) = r$ **then**
14: $\quad\quad \Sigma^{\mu} = \Sigma^{\mu}\theta;\ AD = AD\theta$;
15: $\quad\quad$ **if** $head^{\mu}\theta$ is cyclic **then**
16: $\quad\quad\quad Acyc = false$;
17: **until** $\Sigma^{\mu} = \Sigma^{\mu}_{old}$
18: **return** $\langle \Sigma^{\mu}, Acyc \rangle$;

Fig. 4.6: Algorithm $Adn^{\exists}$

dependencies Σ^{μ} along with a boolean value $Acyc$. As mentioned before, the aim of the algorithm is twofold: it defines a termination criterion on its own, and it can be combined with other termination criteria.

More specifically, if $Acyc$ is *false*, then a form of cyclicity has been detected; otherwise, for every database D, there is a terminating standard Chase sequence of D with Σ.

As for the second aim of the algorithm, the adorned set of dependencies Σ^{μ} given as output can be used as follows: a sufficient condition for checking membership in $\mathsf{CT}^{\mathsf{std}}_{\exists}$ is applied to Σ^{μ} rather than Σ. If Σ^{μ} satisfies the condition, then the original set of dependencies Σ is in $\mathsf{CT}^{\mathsf{std}}_{\exists}$.

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

The algorithm maintains two sets Σ^{μ} and AD, containing the adorned dependencies and the adornment definitions currently derived, respectively. These two sets are also used by Function *adorn*, which is called by Algorithm $Adn^{\exists}$ to verify whether a dependency r can be adorned, on the basis of Σ^{μ} and AD (these are not explicitly passed to Function *adorn*, but are treated as "global variables"). Specifically, to see if a dependency $r = body \rightarrow head$

can be adorned, function *adorn* proceeds as follows. It checks if there is a coherent adorned version $body^\mu$ of *body* (obtained using adorned predicates in $AP(\Sigma^\mu)$) such that there is no dependency in Σ^μ having $body^\mu$ as body. If such a coherent adorned version $body^\mu$ exists, the adorned head $head^\mu = HeadAdn(r, body^\mu,\ AD(\Sigma^\mu))$ is computed, by propagating adornments from $body^\mu$. If $r^\mu = body^\mu \rightarrow head^\mu$ is fireable w.r.t. Σ^μ, then r^μ can be added to Σ^μ, and thus is returned along with the boolean value *true*. Otherwise, the input dependency r is returned along with the boolean value *false*.

We now go into the details of Algorithm $Adn^\exists$. Initially, *Acyc* is *true*, AD is empty, and Σ^μ contains a dependency $p(X_1, ..., X_n) \rightarrow p^{b...b}(X_1, ..., X_n)$ for each $p \in \mathcal{R}$ (lines 1–3). As the algorithm proceeds, Σ^μ and AD are extended and modified; in the case a form of cyclicity is detected the value of *Acyc* is changed to *false*. Specifically, the algorithm proceeds as follows (until Σ^μ does not change).

It first checks if there is a universally quantified dependency r that can be adorned (line 6), using function *adorn*. If this is the case, the corresponding adorned dependency r^μ is added to Σ^μ (line 7). Moreover, if r is an EGD and is not satisfied by $D^\mu(\Sigma^\mu)$, then the *ChaseStep* function executes a Chase step over $D^\mu(\Sigma^\mu)$ with r (line 9). Notice that facts in $D^\mu(\Sigma^\mu)$ contains bound (i.e., b's) and free (i.e., f_i's) symbols: the former is treated as a constant while the latter are treated as labeled nulls. If the Chase step replaces f_i with s, where s is either b or an f_j with $i \neq j$, then $\tau = \{f_i/s\}$.[2] Finally, τ is applied to Σ^μ, all the definitions of f_i are deleted from AD, and τ is applied to AD—to replace occurrences of f_i in the right-hand side of adornment definitions (line 10).

When there are no full dependencies that can be adorned, the algorithm checks if there is an existentially quantified dependency that can be adorned (line 11), and if so, a corresponding adorned dependency r^μ is added to Σ^μ (line 12).

After a dependency is adorned into r^μ, the algorithm checks if there exists a non-empty valid substitution θ s.t. $r^\mu\theta$ is equal to r^υ for some r^υ in Σ^μ (line 13). If this is the case, then θ is applied to Σ^μ and AD (line 14). This ensures termination of $Adn^\exists$. Moreover, if $head^\mu\theta$ is cyclic, then a form of cyclicity that may lead to non-termination is detected and *Acyc* is set to *false* (line 16).

[2] With a slight abuse of notation, here we allow adornment substitutions containing f_i/b.

Input: Dependency $r = body \rightarrow head$.
Output: Pair $\langle bool, r' \rangle$, where $bool$ is a Boolean value and r' is a possibly adorned dependency.
1: **if** $\exists body^{\mu} \in \mathcal{A}(body, AP(\Sigma^{\mu}))$ s.t.
  a) $body^{\mu}$ is coherent,
  b) $body^{\mu} \notin B^{\mu}(r, \Sigma^{\mu})$, and
  c) $r^{\mu} = body^{\mu} \rightarrow head^{\mu}$ is fireable w.r.t. Σ^{μ},
  where $head^{\mu} = HeadAdn(r, body^{\mu}, AD)$ **then**
2:   **return** $\langle true, r^{\mu} \rangle$;
3: **else**
4:   **return** $\langle false, r \rangle$;

Fig. 4.7: Function *adorn*

The overall process described so far is iterated as long as Σ^{μ} changes.

Example 4.58. Consider the set of dependencies $\Sigma_{1.2}$ of Example 1.2. Initially, the following two adorned dependencies, mapping unadorned atoms to atoms adorned with strings of b's, are added to $\Sigma^{\mu}_{1.2}$:

$$s_1 : n(X) \quad \rightarrow n^b(X)$$
$$s_2 : e(X,Y) \rightarrow e^{bb}(X,Y)$$

The algorithm then proceeds by adorning full dependencies and adds the following adorned dependencies to $\Sigma^{\mu}_{1.2}$:

$$s_3 : e^{bb}(X,Y) \rightarrow X = Y$$
$$s_4 : e^{bb}(X,Y) \rightarrow n^b(X)$$

Notice that $D^{\mu}(\Sigma^{\mu}_{1.2}) = \{n(b), e(b,b)\}$ and thus the EGD r_3 in $\Sigma_{1.2}$ is satisfied by $D^{\mu}(\Sigma^{\mu}_{1.2})$. Next, the existentially quantified dependency (namely, r_1) is adorned and the following adorned dependency is added to $\Sigma^{\mu}_{1.2}$:

$$s_5 : n^b(X) \rightarrow \exists Y \; e^{bf_1}(X,Y)$$

Moreover, $AD = \{f_1 = f^{r_1}_{Y}(b)\}$. After that, the algorithm starts considering full dependencies again. By adorning the EGD r_3, the following adorned dependency is obtained, which is added to $\Sigma^{\mu}_{1.2}$:

$$s_6 : e^{bf_1}(X,Y) \rightarrow X = Y$$

Notice that $D^{\mu}(\Sigma^{\mu}_{1.2}) = \{n(b), e(b,b), e(b,f_1)\}$ and thus $D^{\mu}(\Sigma^{\mu}_{1.2}) \not\models r_3$. Thus, function *ChaseStep* is executed with $D^{\mu}(\Sigma^{\mu}_{1.2})$ and r_3, returning the substitution $\theta = \{f_1/b\}$, which is applied to $\Sigma^{\mu}_{1.2}$, whereas AD becomes empty.

After the application of θ, we have that $\Sigma^{\mu}_{1.2} = \{s_1, s_2, s_3, s_4, s'_5\}$, where s'_5 is derived from s_5 by replacing f_1 with b, that is, $s'_5 :\ n^b(X) \rightarrow \exists Y\ E^{bb}(X,Y)$.

At this point, no further dependencies can be adorned and the algorithm terminates by returning the value $Acyc = true$ along with $\Sigma^{\mu}_{1.2}$. Notice that there is no dependency (of any kind) that can be adorned because $AP(\Sigma^{\mu}_{1.2}) = \{n^b, e^{bb}\}$ and the body of the dependencies in $\Sigma_{1.2}$ have already been adorned using these adorned predicates. □

Example 4.59. Consider the set of dependencies $\Sigma_{4.51}$ of Example 4.51. Initially, the following adorned dependencies are added to $\Sigma^{\mu}_{4.51}$:

$$s_1 :\ n(X) \rightarrow n^b(X)$$
$$s_2 :\ e(X,Y,Z) \rightarrow e^{bbb}(X,Y,Z)$$

Then, full dependencies are adorned and the following adorned dependencies are added to $\Sigma^{\mu}_{4.51}$:

$$s_3 :\ e^{bbb}(X,Y,Z) \rightarrow Y = Z$$
$$s_4 :\ e^{bbb}(X,Y,Y) \rightarrow n^b(Y)$$

Notice that $D^{\mu}(\Sigma^{\mu}_{4.51}) = \{n(b), e(b,b,b)\}$ and thus the EGD r_3 in $\Sigma_{4.51}$ is satisfied by $D^{\mu}(\Sigma^{\mu}_{4.51})$. Next, the existentially quantified dependency (namely, r_1) is adorned and the following adorned dependency is added to $\Sigma^{\mu}_{4.51}$:

$$s_5 :\ n^b(X) \rightarrow \exists Y\ \exists Z\ e^{bf_1f_2}(X,Y,Z)$$

with $AD = \{f_1 = f^{r_1}_Y(b), f_2 = f^{r_1}_Z(b)\}$. Now universally quantified dependencies are considered again to see if they can be adorned. Suppose r_3 is chosen. Then, the following adorned dependency is added to $\Sigma^{\mu}_{4.51}$:

$$s_6 :\ e^{bf_1f_2}(X,Y,Z) \rightarrow Y = Z$$

Now, $D^{\mu}(\Sigma^{\mu}_{4.51}) = \{n(b), e(b,b,b), e(b,f_1,f_2)\}$, which does not satisfy the EGD r_3. By executing the *ChaseStep* function on $D^{\mu}(\Sigma^{\mu}_{4.51})$ and r_3, the substitution $\tau = \{f_2/f_1\}$ is obtained (alternatively, f_1/f_2 might have been chosen, but the choice is immaterial). Then, the adornment definition $f_2 = f^{r_1}_Z(b)$ is removed from AD, and the substitution τ is applied to both $\Sigma^{\mu}_{4.51}$ and AD, replacing f_2 with f_1. Thus, AD becomes $\{f_1 = f^{r_1}_Y(b)\}$, while s_5 and s_6 become:

$$s'_5: \ n^b(X) \rightarrow \exists Y \ \exists Z \ e^{bf_1f_1}(X,Y,Z)$$
$$s'_6: \ e^{bf_1f_1}(X,Y,Z) \rightarrow Y = Z$$

By proceeding as discussed above, the following adorned dependencies are added to $\Sigma^{\mu}_{4.51}$:

$$\begin{aligned}
s_7 &: \ e^{bf_1f_1}(X,Y,Y) \rightarrow n^{f_1}(Y) \\
s_8 &: \ n^{f_1}(X) \rightarrow \exists Y \ \exists Z \ e^{f_1f_3f_3}(X,Y,Z) \\
s_9 &: \ e^{f_1f_3f_3}(X,Y,Z) \rightarrow Y = Z \\
s_{10} &: \ e^{f_1f_3f_3}(X,Y,Y) \rightarrow n^{f_3}(Y) \\
s_{11} &: \ n^{f_3}(X) \rightarrow \exists Y \ \exists Z \ e^{f_3f_5f_5}(X,Y,Z) \\
s_{12} &: \ e^{f_3f_5f_5}(X,Y,Z) \rightarrow Y = Z \\
s_{13} &: \ e^{f_3f_5f_5}(X,Y,Z) \rightarrow n^{f_5}(Y) \\
s_{14} &: \ n^{f_5}(X) \rightarrow \exists Y \ \exists z \ e^{f_5f_7f_7}(X,Y,Z) \\
s_{15} &: \ e^{f_5f_7f_7}(X,Y,Z) \rightarrow Y = Z
\end{aligned}$$

with $AD = \{f_1 = f^{r_1}_Y(b), \ f_3 = f^{r_1}_Y(f_1), \ f_5 = f^{r_1}_Y(f_3), \ f_7 = f^{r_1}_Y(f_5)\}$. When s_{15} is introduced, a valid substitution $\theta = \{f_5/f_1, f_7/f_3\}$ mapping s_{15} to s_9 is found. Thus, θ is applied to both $\Sigma^{\mu}_{4.51}$ and AD, replacing all occurrences of adornment symbols f_5 and f_7 with f_1 and f_3, respectively. Notice that dependencies $s_{11} - s_{14}$ become:

$$\begin{aligned}
s'_{11} &: \ n^{f_3}(X) \rightarrow \exists y \ \exists z \ e^{f_3f_1f_1}(X,Y,Z) \\
s'_{12} &: \ e^{f_3f_1f_1}(X,Y,Z) \rightarrow Y = Z \\
s'_{13} &: \ e^{f_3f_1f_1}(X,Y,Y) \rightarrow n^{f_1}(Y) \\
s'_{14} &: \ n^{f_1}(X) \rightarrow \exists Y \ \exists z \ e^{f_1f_3f_3}(X,Y,Z)
\end{aligned}$$

while s_{15} becomes equal to s_9. Moreover, $AD = \{f_1 = f^{r_1}_Y(b), \ f_3 = f^{r_1}_Y(f_1), \ f_1 = f^{r_1}_Y(f_3)\}$. Since $\Omega(\Sigma^{\mu}_{4.51})$ is cyclic (after the application of θ), as it contains the edges (f_1, f_3) and (f_3, f_1), variable *Acyc* is set to *false*.

At this point, no further dependencies can be adorned and the algorithm terminates by returning the value $Acyc = false$ along with $\Sigma^{\mu}_{4.51}$. □

Theorem 4.60. *Algorithm* $Adn^{\exists}$ *terminates for every set of dependencies.* □

Thus, given an input set of dependencies Σ, Algorithm $Adn^{\exists}$ always returns a pair consisting of a set Σ^{μ} of adorned dependencies and a boolean value *Acyc* giving information about the detection of a form of cyclicity—we use $Adn^{\exists}(\Sigma)[1]$ to refer to Σ^{μ} and $Adn^{\exists}(\Sigma)[2]$ to refer to *Acyc*.

$\|\Sigma_{egd}\|$ / $\|\Sigma_\exists\|$	$[1, 10]$		$[11, 100]$	
	#tests	$\|\Sigma\|$	#tests	$\|\Sigma\|$
[1, 10]	50	86	7	451
[11, 100]	15	406	26	1,210
[101, 1000]	51	3,113	13	3,176
[1001, 5000]	9	9,117	7	19,587

(a) Ontologies' Size

$[1, 10]$		$[11, 100]$	
$\|\Sigma^\mu\|/\|\Sigma\|$	Time	$\|\Sigma^\mu\|/\|\Sigma\|$	Time
2.38	84	3.15	125
2.45	141	2.83	275
2.97	787	6.16	22,819
2.82	712	2.82	1,495

(b) Complexity

$[1, 10]$		$[11, 100]$	
A+NT	FN	A+NT	FN
50$_{[44+6]}$	**0**	**7**$_{[6+1]}$	**0**
15$_{[6+9]}$	**0**	**26**$_{[13+13]}$	**0**
51$_{[4+47]}$	**0**	**11**$_{[1+10]}$	**2**
9$_{[0+9]}$	**0**	**7**$_{[0+7]}$	**0**

(c) Expressivity

Table 4.2: Experimental Results.

Another important property of Algorithm $Adn^{\exists}$ is stated in the next theorem. It says that, given a set of dependencies Σ and a database D, some of the canonical models of (D, Σ) can be obtained from the canonical models of (D, Σ^μ) by dropping adornments, where $\Sigma^\mu = Adn^{\exists}(\Sigma)[1]$. Moreover, whenever (D, Σ) has canonical models, (D, Σ^μ) admits canonical models as well. These two properties imply that if (D, Σ) has canonical models, then we can construct one from (D, Σ^μ) (e.g., by using the core Chase).

Theorem 4.61. *Consider a set of dependencies Σ and let $\Sigma^\mu = Adn^{\exists}(\Sigma)[1]$. For every database D,*

1. *$src(CMod(D, \Sigma^\mu)) \subseteq CMod(D, \Sigma)$, and*
2. *$CMod(D, \Sigma^\mu) \neq \varnothing$ iff $CMod(D, \Sigma) \neq \varnothing$.* □

On the basis of the boolean value returned by Algorithm $Adn^{\exists}$, below we define *semi-acyclic* dependencies.

Definition 4.62 (Semi-acyclic dependencies). *A set of dependencies Σ is* semi-acyclic (SAC) *if $Adn^{\exists}(\Sigma)[2]$ is true.* □

Every semi-acyclic set of dependencies belongs to $\mathsf{CT}_{\exists}^{\mathsf{std}}$.

Theorem 4.63. *For every semi-acyclic set of dependencies Σ and for every database D, there is a terminating standard Chase sequence of D with Σ whose length is polynomial in the size of D.* □

4.4.3 Expressivity, Complexity and Experimental Evaluation

As Algorithm $Adn^{\exists}$ embeds the fireable condition of semi-stratification, we have that semi-acyclicity strictly generalizes semi-stratification. It also generalizes acyclicity.

Theorem 4.64. *$\mathcal{S}$-$\mathcal{S}tr \subsetneq \mathcal{SAC}$ and $\mathcal{AC} \subsetneq \mathcal{SAC}$.* □

As $\mathcal{SAC}$ includes sets of dependencies which are not in $\mathsf{CT}^{\mathsf{std}}_{\forall}$, it follows that $\mathcal{SAC} \not\subseteq \mathcal{MFA}$; it is an open problem whether $\mathcal{MFA} \subseteq \mathcal{SAC}$.

We now turn our attention to the second aim of Algorithm $Adn^{\exists}$: providing a set of adorned dependencies Σ^{μ} which can be used in place of the original set of dependencies Σ for termination analysis. As shown in the following, Σ^{μ} turns out to be better than Σ for the purpose of checking termination (see Theorem 4.66 below).

Given a termination criterion C, we use $Adn^{\exists}$-$\mathcal{C}$ to denote the class of sets of dependencies Σ such that $Adn^{\exists}(\Sigma)[1]$ belongs to $\mathcal{C}$. Moreover, we define $\mathbb{C}$ as the set containing $\mathcal{C}$ for every criterion C discussed in Section 4.2.

The following theorem states that by combining Algorithm $Adn^{\exists}$ with current termination criteria (including those for checking if a set of dependencies belongs to $\mathsf{CT}^{\mathsf{std}}_{\forall}$), we can check (via a sufficient condition) if a set of dependencies belongs to $\mathsf{CT}^{\mathsf{std}}_{\exists}$. Theorem 4.66 below says that by proceeding in this way we can identify strictly more sets of dependencies in $\mathsf{CT}^{\mathsf{std}}_{\exists}$.

Theorem 4.65. *Let Σ be a set of dependencies. If $\Sigma \in Adn^{\exists}$-$\mathcal{C}$ then $\Sigma \in \mathsf{CT}^{\mathsf{std}}_{\exists}$, for $\mathcal{C} \in \mathbb{C}$.* □

Theorem 4.66. *$\mathcal{C} \subsetneq Adn^{\exists}$-$\mathcal{C}$, for $\mathcal{C} \in \mathbb{C}$.* □

The previous theorem follows from the fact that if a set of dependencies satisfies a termination condition, then its adorned version has the same (or weaker) structural properties and thus it satisfies the termination condition too.

We point out that if $\Sigma \in Adn^{\exists}$-$\mathcal{C}$ then $\Sigma \in \mathsf{CT}^{\mathsf{std}}_{\exists}$, but it can be the case that $\Sigma \notin \mathsf{CT}^{\mathsf{std}}_{\forall}$ even if C is a criterion for checking if a set of dependencies is in $\mathsf{CT}^{\mathsf{std}}_{\forall}$.

The following theorem states the complexity of Algorithm $Adn^{\exists}$.

Theorem 4.67. *For any set of dependencies Σ, the size of $Adn^{\exists}(\Sigma)[1]$ and the time complexity of computing it using Algorithm $Adn^{\exists}$ are exponential and double exponential in the size of Σ, respectively.* □

Despite of the theorem above, as shown in our experimental evaluation, the size of Σ^{μ} and the time to compute it are reasonable in practice.

Experimental Evaluation. We now report on an experimental evaluation we performed to assess our approach. We have implemented Algorithm $Adn^{\exists}$ in Java. The implementation, as well as the datasets we used, can be found at `http://si.deis.unical.it/~calautti/chase/`. We used sets of dependencies taken from the repository [1], which includes ontologies in a variety of domains: a large subset of the Gardiner ontology corpus [42], the LUBM ontology [57], several Phenoscape ontologies [3], and a number of ontologies from two versions of the Open Biomedical Ontology corpus [2]. All experiments were run on an Intel i7-3770 3.40 Ghz, 16 GB of memory.

Table 4.2 resumes (a) the main characteristics of the dependency sets used in our experiments, (b) the complexity of analyzing a set of dependencies in terms of the number of generated adorned rules and the time to compute them, and (c) the expressive power in terms of the number of sets of dependencies recognized as terminating or not.

More specifically, we considered a collection of 178 ontologies and partitioned it into eight classes depending on the number of existentially quantified TGDs and the number of EGDs. For the former we considered four intervals, namely $[1, 10]$, $[11, 100]$, $[101, 1000]$ and $[1001, 5000]$, while for the latter we considered two intervals, namely $[1, 10]$ and $[11, 100]$. For each class, we have considered ontologies with different ratios $|\Sigma_{\forall}|/|\Sigma_{\exists}|$.

Table 4.2a reports, for each class, the number of ontologies belonging to the class (column #tests) along with the average number of dependencies for the ontologies in the class (column $|\Sigma|$).

Table 4.2b shows, for each class, the average ratio of the number of adorned dependencies to the number of dependencies in the original ontology (column $|\Sigma^{\mu}|/|\Sigma|$), along with the average time (in milliseconds) to compute the adorned set (column Time). It is worth noting that the set of adorned dependencies in not much larger than the original set of dependencies, and running times are lower than 1 second in most of the cases.

Table 4.2c reports, for each class, *(i)* the number of semi-acyclic ontologies + the number of ontologies that are not semi-acyclic and the standard Chase did not halt within 24 hours (column $A+NT$), and *(ii)* the number of ontologies that are not semi-acyclic and the standard Chase terminated within 24 hours (column FN, "false negatives"). Notice that, among the 76 ontologies for which the Chase terminated, only 2 were not semi-acyclic.

5

Conclusions

The termination of programs has proven to be an appealing and challenging problem for the Computer Science community, and has seen a very important advancement in the field of logic programming and database integrity constraints satisfaction. In particular, the Chase has proven to be a central tool in many current applications. Its importance is due to the fact that several problems can be solved by exhibiting a universal model, and the Chase computes a universal model, when it terminates.

This thesis studied the termination problem for logic programming with function symbols and for database dependencies by presenting the state of the art and discussed new approaches in these fields with the aim to overcome the limitations of current works.

For logic programs with function symbols, a new approach that exploits the presence of different function symbols in order to understand whether a logic program terminates was discussed, along with another approach that characterizes the termination of logic programs via acyclicity notions over particular graph constructed from the given programs. Finally, multiple approaches based on linear constraints solving have been proposed.

For the case of database dependencies, it has been shown that the (semi-)oblivious Chase termination problem for guarded-based TGDs is decidable, and precise complexity bounds are obtained as well. Furthermore, a novel approach for directly dealing with the ∃-sequence Chase termination problem in the presence of both TGDs and EGDs is discussed.

Interesting directions for future work regarding the termination of logic programs are to plug current approaches in the framework proposed in [34]

and study their combination in such a framework, and analyze the relationships between the notions of safety of [34] and the notions of limitedness of termination techniques.

Desirable advancements in the context of Chase termination are to close the picture for guarded-based TGDs by also considering the standard Chase, provide similar results for other well-known classes of TGDs, like sticky TGDs [25] and study a new adornment algorithm for dealing with the $\forall$-sequence Chase termination problem in the presence of both TGDs and EGDs.

6

Appendix

Proofs of Section 4.3

Proof of Lemma 4.16

By hypothesis, there exists a cycle in $EDG(\Sigma)$ that contains a special edge; let $(v_0, v_1), (v_1, v_2), \ldots, (v_{n-1}, v_n)$ be such a cycle $(v_0 = v_n)$ with $\lambda((v_i, v_{i+1})) = (r_i, k_i)$, for each $0 \leqslant i < n$. Assuming that the above cycle is one of the shortest cycles in $EDG(\Sigma)$ that contains a special edge, we can show that there exist sequences $I_0, I_1, \ldots$ and $(\rho_0, h_0), (\rho_1, h_1), \ldots$, where, for $k \in \mathbb{N}_0$, $k \cdot n \leqslant i < (k+1) \cdot n$ implies $\rho_i = r_{i-k \cdot n}$, such that:

1. for each $i \geqslant 0$, $I_i \langle \rho_i, h_i \rangle I_{i+1}$; and
2. for each $i \neq j \geqslant 0$, $\rho_i = \rho_j$ implies $h_i \neq h_j$.

This immediately implies that Σ admits an infinite obl-chase derivation, as needed. The proof for the existence of the above sequences is by induction on $i \geqslant 0$.

Base Step. Let $I_0 = \{p(\mathbf{t})\}$, where p/n is the predicate of $body(r_0)$ and $\mathbf{t} = \{c\}^n$, where $c \in \mathbf{C}$. Clearly, the homomorphism $h_0 = \{X \rightarrow c \mid X \in var(body(r_0))\}$ is such that $h_0(body(r_0)) \in I_0$. Since $\rho_0 = r_0$, we conclude that (ρ_0, h_0) is a trigger for Σ on I_0, and claim (1) follows. Since $I_0 \langle \rho_0, h_0 \rangle I_1$ involves only one trigger, claim (2) holds trivially.

Inductive Step. By induction hypothesis, (ρ_i, h_i) is a trigger for Σ on I_i, and $I_i \langle \rho_i, h_i \rangle I_{i+1}$, where $I_{i+1} = I_i \cup \{h'_i(head(\rho_i))\}$ with $h'_i \supseteq h_i$. Observe that on π there exist edges (v, u) and (u, w) such that $\lambda((v, u)) = (\rho_i, k)$ and

$\lambda((u,w)) = (\rho_{i+1}, k')$. This implies that $pred(A_i^k) = pred(body(\rho_{i+1}))$, where A_i^k is the k-th atom of $head(\rho_i)$. Since ρ_{i+1} is a simple linear TGD, there exists a homomorphism μ such that $\mu(body(\rho_{i+1})) = h'_i(A_i^k)$. Since $h'_i(A_i^k) \in I_{i+1}$, with $h_{i+1} = \mu$, (ρ_{i+1}, h_{i+1}) is a trigger for Σ on I_{i+1}, and claim (1) follows.

We proceed to establish claim (2). By induction hypothesis, it suffices to show that, for each $0 \leqslant j \leqslant i$, $\rho_j = \rho_{i+1}$ implies $h_j \neq h_{i+1}$. Assuming that $\rho_{i+1} = r_{(i+1)-k\cdot n}$, for some $k \in \mathbb{N}_0$, we consider the cases where $0 \leqslant j \leqslant k \cdot n$ and $k \cdot n < j \leqslant i$.

Case 1. Assume first that $0 \leqslant j \leqslant k \cdot n$. For each $0 \leqslant j \leqslant k \cdot n$ such that $\rho_j \neq \rho_{i+1}$, the claim follows immediately. Consider now an arbitrary $j \in \{0, \ldots, k \cdot n\}$ such that $\rho_j = \rho_{i+1}$. Due to the occurrence of a special edge in π — w.l.o.g., we assume that is the first edge of π — we can conclude that h_{i+1} maps $var(body(\rho_{i+1}), \{u\})$ to a null $z \in \mathbf{N}$ that was invented during or after the trigger application $I_{k\cdot n}\langle \rho_{k\cdot n}, h_{k\cdot n}\rangle I_{k\cdot n+1}$. Therefore, z does not occur in I_j, which in turn implies that h_j maps $var(body(\rho_j), \{u\})$ to a term other than z. Thus, $h_j \neq h_{i+1}$, and the claim follows.

Case 2. Towards a contradiction, assume that there exists $j \in \{k \cdot n + 1, \ldots, i\}$ such that $\rho_j = \rho_{i+1}$ and $h_j = h_{i+1}$, i.e., $(\rho_j, h_j) = (\rho_{i+1}, h_{i+1})$. Thus, the application of the trigger (ρ_{i+1}, h_{i+1}) can be avoided, and obtain a shorter Chase sequence. This implies that in $EDG(\Sigma)$ there exists a cycle that contains a special edge with length less that n. But this contradicts the fact that $(v_0, v_1), (v_1, v_2), \ldots, (v_{n-1}, v_n)$ is one of the shortest cycles that contains a special edge, and the claim follows. □

Proof of Lemma 4.27

Let us first show that <u>r^{i+2} is active</u>. Towards a contradiction, assume that $\mathsf{R}(r^{i+2}) = \bot$. This implies that $\mathsf{R}(r^{i+1})$ is not compatible with r. By definition of compatibility, we conclude that there exists a variable $X \in var(body(r))$, with $\Pi = pos(body(r), \{X\})$, such that there exist two distinct variables $Z, W \in var(head(\mathsf{R}(r^{i+1})), \Pi)$, and at least one of them is an existentially quantified variable; assume that $Z \in ex(\mathsf{R}(r^{i+1}),)$. For technical clarity, assume that $|\Pi| = 2$; our argument can be extended to the general case. Let $\pi_Z = pos(head(\mathsf{R}(r^{i+1})), \{Z\}) \cap \Pi$ and $\pi_W = pos(head(\mathsf{R}(r^{i+1})), \{W\}) \cap \Pi$. Since $Z \neq W$, for each $j < i+1$, $var(head(\mathsf{R}(r^j)), \{\pi_Z\}) \in fr(\mathsf{R}(r^j))$; otherwise, $\mathsf{R}(r^{i+1}) = \bot$, which contradicts our hypothesis. Regarding the position π_W, we consider the two cases where $var(head(\mathsf{R}(r^{i+1})), \{\pi_W\})$ either belongs to $fr(\mathsf{R}(r^{i+1}))$, or to $ex(\mathsf{R}(r^{i+1}),)$. In both cases, for each $j < i+1$,

$var(head(\mathsf{R}(r^j)), \{\pi_W\}) \in fr(\mathsf{R}(r^j))$. From the above discussion, we conclude that, during the construction of $\mathsf{R}(\mathsf{R}(r^i), r)$, the two distinct frontier variables $var(head(\mathsf{R}(r^i)), \{\pi_Z\})$ and $var(head(\mathsf{R}(r^i)), \{\pi_W\})$ must be unified, which in turn implies that $eqtype(\mathsf{R}(r^i)) \neq eqtype(\mathsf{R}(r^{i+1}))$. But this contradicts our hypothesis, and therefore r^{i+2} is active, as needed.

It remains to show that $\underline{eqtype(\mathsf{R}(r^{i+1})) = eqtype(\mathsf{R}(r^{i+2}))}$. First, observe that $eqtype(\mathsf{R}(r^{i+1})) \subseteq eqtype(\mathsf{R}(r^{i+2}))$, since, during the construction of $\mathsf{R}(r^{i+2})$, we only unify symbols in $body(\mathsf{R}(r^{i+1}))$, and thus add equalities among positions to $eqtype(\mathsf{R}(r^{i+1}))$. We proceed to show the other direction. Towards a contradiction, assume that $eqtype(\mathsf{R}(r^{i+2})) \not\subseteq eqtype(\mathsf{R}(r^{i+1}))$. Since $eqtype(\mathsf{R}(r^{i+1})) \subseteq eqtype(\mathsf{R}(r^{i+2}))$, we get that $eqtype(\mathsf{R}(r^{i+1})) \subsetneq eqtype(\mathsf{R}(r^{i+2}))$. This implies that there exist two positions π_1 and π_2 in $head(\mathsf{R}(r^{i+1}))$ such that $var(head(\mathsf{R}(r^{i+1})), \{\pi_1\}) = X \neq var(head(\mathsf{R}(r^{i+1})), \{\pi_2\}) = Y$, and also $X, Y \in fr(\mathsf{R}(r^{i+1}))$. Therefore, during the construction of $\mathsf{R}(\mathsf{R}(r^{i+1}), r)$, X and Y must be unified. Hence, the same positions π_1 and π_2 in $head(\mathsf{R}(r^i))$ also contain two distinct frontier variables of $fr(\mathsf{R}(r^i))$, that were forced to unify during the construction of $\mathsf{R}(\mathsf{R}(r^i), r)$. Consequently, $eqtype(\mathsf{R}(r^i)) \neq eqtype(\mathsf{R}(r^{i+1}))$. But this contradicts our hypothesis, and therefore $eqtype(\mathsf{R}(r^{i+1})) = eqtype(\mathsf{R}(r^{i+2}))$, as needed. This completes our proof. □

Proof of Lemma 4.29

The proof is along the lines of the proof for Lemma 4.16, where an analogous result is established for simple linear TGDs. However, in the inductive step of the proof for Lemma 4.16, we rely on the fact that the given set of linear TGDs is simple. More precisely, due to the fact that ρ_{i+1} is simple linear, we immediately conclude that there exists a homomorphism μ such that $\mu(body(\rho_{i+1})) = h'_i(A_k)$ (see the third line of the inductive step in the proof of Lemma 4.16). However, if ρ_{i+1} is a (non-simple) linear TGD, then the existence of μ is not immediate; in fact, to show that μ exists is a non-trivial task. The rest of the proof is devoted to establish the existence of μ. Before we proceed further, let us first show the following auxiliary lemma, which, roughly speaking, states that the resolvent of an active sequence of single-head linear TGDs mimics the behavior of the sequence during the Chase.

Lemma 6.1. *Let $I_0 = \{p(c, \ldots, c)\}, I_1, \ldots, I_n$, where $n > 0$, be a sequence of instances such that $I_i\langle r_i, h_i\rangle I_{i+1}$, for each $0 \leqslant i < n$, with $r_0, \ldots, r_{n-1}$ be*

an active sequence of single-head linear TGDs. Moreover, we assume that, for every $1 \leqslant i \leqslant n-1$, $h_i(body(r_i)) \in (I_i \setminus I_{i-1})$[1]*. The atom obtained by applying* (r_{n-1}, h_{n-1}) *to* I_{n-1} *coincides (modulo null renaming) with the atom obtained by applying* (ρ, μ) *to* I_0*, where* $\rho = \mathsf{R}(r_0, \ldots, r_{n-1})$ *and* $\mu = \{X \to c \mid X \in var(body(\rho))\}$*.*

Proof. The proof is by induction on $n > 0$.

Base Step. The claim holds trivially, since $r_0 = \mathsf{R}(r_0) = \rho$.

Inductive Step. By induction hypothesis, the atom obtained by applying (r_{n-2}, h_{n-2}) to I_{n-2} coincides, modulo null renaming, with the atom obtained by applying $(\hat{\rho}, g)$ to I_0, where $\hat{\rho} = \mathsf{R}(r_0, \ldots, r_{n-2})$ and $g = \{X \to c \mid X \in var(body(\hat{\rho}))\}$. Therefore, $h_{n-1}(body(r_{n-1})) = g'(head(\hat{\rho}))$, where $g' \supseteq g$ maps each variable $X \in ex(\hat{\rho},)$ to a "fresh" null. By construction, ρ is the TGD $\theta(body(\hat{\rho})) \to \theta(head(r_{n-1}))$. Assuming that $h'_{n-1}(head(r_{n-1}))$, where $h'_{n-1} \supseteq h_{n-1}$, is the atom obtained by applying (r_{n-1}, h_{n-1}) to I_{n-1}, it is clear that $(g' \cup h'_{n-1})$ is a unifier for $head(\hat{\rho})$ and $body(r_{n-1})$. By definition of the most general unifier, there exists a substitution λ such that $(\lambda \circ \theta) = (g' \cup h'_{n-1})$. Observe that

$$\lambda(body(\rho)) = \lambda(\theta(body(\hat{\rho}))) = (g' \cup h'_{n-1})(body(\hat{\rho})) = g(body(\hat{\rho})) = p(c, \ldots, c),$$

and

$$\lambda(head(\rho)) = \lambda(\theta(head(r_{n-1}))) = (g' \cup h'_{n-1})(head(r_{n-1})) = h'_{n-1}(head(r_{n-1})).$$

Since $\lambda_{|fr(\rho)} = \mu$, the claim follows.

Let us now establish the existence of μ. By hypothesis, there exists a critical cycle in $EDG(\Sigma)$ that contains a special edge; let $\pi = (v_0, v_1), (v_1, v_2), \ldots, (v_{n-1}, v_n)$ be such a cycle ($v_0 = v_n$) with $\lambda((v_i, v_{i+1})) = (r_i, k_i)$, for each $0 \leqslant i < n$. Since $(r_0, k_0), \ldots, (r_{n-1}, k_{n-1})$ is a critical sequence, Lemma 4.28 implies that the sequence $(\rho_0, j_0), (\rho_1, j_1), \ldots, (\rho_i, j_i)$, where, for $k \in \mathbb{N}_0$, $k \cdot n \leqslant \ell < (k+1) \cdot n$ implies $\rho_\ell = r_{i-k \cdot n}$, is active, i.e., $\mathsf{R}((\rho_0, j_0), (\rho_1, j_1), \ldots, (\rho_i, j_i)) \neq \bot$; the $j_i's$ refer to the head-atoms of $\rho_i's$ that appear on the critical cycle. By Lemma 6.1, it suffices to show that there exists a homomorphism that maps $body(\rho_{i+1})$ to the atom obtained by applying (τ, g) to $\{p(c, \ldots, c)\}$, where $\tau = \mathsf{R}((\rho_0, j_0), (\rho_1, j_1), \ldots, (\rho_i, j_i))$ and

[1] This additional assumption simply says that the TGD r_i is triggered by the atom obtained after applying r_{i-1}.

$g = \{X \to c \mid X \in \mathit{var}(\mathit{body}(\tau))\}$. Let μ be the substitution that maps the variables of $\mathit{var}(\mathit{body}(\rho_{i+1}), \Pi)$, where Π are the frontier positions of τ, to the constant c, and all the other variables of $\mathit{var}(\mathit{body}(\rho_{i+1}))$ to nulls, according to the atom obtained after the application of (τ, g) to $\{p(c, \ldots, c)\}$. The fact that τ is compatible with ρ_{i+1} implies that μ is well-defined, and the claim follows. □

Proof of Lemma 4.37

Consider a directed graph $G = (N, E)$, and two nodes $s, t \in N$. We are going to construct a database D, a set $\Sigma \in (\mathsf{SL} \cap \mathsf{CT}^\star)$, where $\star \in \{\mathsf{obl}, \mathsf{sobl}\}$, and a propositional predicate q, such that $D \cup \Sigma \models q$ iff t is reachable from s. The idea is to construct Σ in such a way that its predicate graph coincides with G, while D stores the node s, and q represents the node t. More precisely,

$$D \;=\; \{p_s(c)\} \quad \text{and} \quad \Sigma \;=\; \{p_v(X) \to p_u(X) \mid (v, u) \in E\} \cup \{p_t(X) \to q\}.$$

It is clear that the above set of simple linear TGDs belongs to $\mathsf{CT}^\star$ since it does not contain existentially quantified variables, and the claim follows. □

Proof of Lemma 4.39

It suffices to show that the complement of our problem is in NSPACE($\omega \log(\omega \cdot |\mathit{sch}(\Sigma)|) + \omega \log(\omega \cdot |\Sigma|)$). To this end, by definition of critical-rich-acyclicity (resp., critical-weak-acyclicity), we need to show that the problem of deciding whether a critical cycle in $\mathit{EDG}(\Sigma)$ (resp., $\mathit{DG}(\Sigma)$) that contains a special edge exists, is in NSPACE($\omega \log(\omega \cdot |\mathit{sch}(\Sigma)|) + \omega \log(\omega \cdot |\Sigma|)$). In the rest of the proof, let $G = (N, E, \lambda)$ be either $\mathit{EDG}(\Sigma)$ or $\mathit{DG}(\Sigma)$.

The problem under consideration can be conceived as an extended version of graph reachability. More precisely, we need to decide whether there exists a node $v \in N$ that is reachable from itself via a cycle $\pi = (v, v_1), (v_1, v_2), \ldots, (v_{n-1}, v)$, and the following hold: (i) π is critical, or, equivalently, $\lambda((v, v_1)), \lambda((v_1, v_2)), \ldots, \lambda((v_{n-1}, v))$ is critical; and (ii) (v, v_1) is special, or (v_{n-1}, v) is special, or (v_i, v_{i+1}) is special, for some $i \in [n-2]$. This can be achieved by applying the following nondeterministic procedure:

1. Guess an edge $e_1 = (v_1, v_2) \in E$.
2. If e_1 is special, then $\mathit{flag} := 1$; otherwise, $\mathit{flag} := 0$.
3. $r_1 := \lambda(e_1)$ and $\mathit{origin} := v_1$.

4. Repeat

   If there is no edge $(u, w) \in E$ such that $u = v_2$, then *reject*; otherwise, guess an edge $e_2 = (v_2, v_3) \in E$.

   If e_2 is special, then $\mathit{flag} := 1$.

   $r_2 := \lambda(e_2)$.

   If r_1 is not compatible with r_2, then *reject*; otherwise, $e_1 = (v_1, v_2) := e_2 = (v_2, v_3)$ and $r_1 := \mathsf{R}(r_1, r_2)$.

   Until $(v_3 = \mathit{origin})$.
5. If $\mathit{flag} = 0$, then *reject*.
6. If $\mathsf{R}\left(\underbrace{r_1, \ldots, r_1}_{k}\right) \neq \bot$, for each $k \in [\omega + 1]$, then *accept*; otherwise, *reject*.

It is not difficult to verify that the above procedure is correct. In fact, the repeat-until statement seeks for an active cycle π in G, and if it exists, the resolvent of the TGDs that label the edges of π is stored in r_1. If such an active cycle does not exist, then the algorithm rejects. Finally, the algorithm returns *accept* iff π contains a special edge (i.e., if $\mathit{flag} = 1$), and π is critical (i.e., the resolvent of the sequence $r_1, \ldots, r_1$ of length k, for each $k \in [\omega + 1]$, exists). The rest of the proof is devoted to show that the above nondeterministic procedure runs in space $O(\omega \log(\omega \cdot |\mathit{sch}(\Sigma)|) + \omega \log(\omega \cdot |\Sigma|))$.

First, observe that encoding a position of $\mathit{sch}(\Sigma)$ requires $\log(\omega \cdot |\mathit{sch}(\Sigma)|)$ space, encoding a predicate of $\mathit{sch}(\Sigma)$ requires $\log(|\mathit{sch}(\Sigma)|)$ space, and encoding a variable occurring in Σ requires $O(\log(\omega \cdot |\Sigma|))$ space – we assume that the TGDs of Σ do not share variables, and thus $O(\omega \cdot |\Sigma|)$ variables may occur in Σ. Therefore, we can maintain an edge of G, that is, a pair of positions, in $O(\log(\omega \cdot |\mathit{sch}(\Sigma)|))$ space, and a single-head linear TGD in $O(\log(|\mathit{sch}(\Sigma)|) + \omega \log(\omega \cdot |\Sigma|))$, that is, the space needed to store two predicates and 2ω variables.

It is easy to verify that, during the execution of the above procedure, we need to remember two edges (e_1 and e_2), two single-head linear TGDs (r_1 and r_2), and some auxiliary values (*flag* and *origin*). Clearly, the above elements can be maintained in $O(\log(\omega \cdot |\mathit{sch}(\Sigma)|) + \omega \log(\omega \cdot |\Sigma|))$ space. However, it should not be forgotten that we need to consider also the space needed to construct $\mathsf{R}(r_1, r_2)$, and to check that, for each $k \in [\omega + 1]$, the sequence $r_1, \ldots, r_1$ of length k is active.

To construct $\mathsf{R}(r_1, r_2)$, we first need to check whether r_1 is compatible with r_2 (see Definition 4.21). This can be easily done in $O(\omega \log(\omega \cdot |\mathit{sch}(\Sigma)|))$ space, which is actually the space needed to store the positions of $\mathit{pos}(\mathit{body}(r_2), \{X\})$,

for some $X \in var(body(r_2))$. Notice that, to check whether $head(r_1)$ and $body(r_2)$ unify, it suffices to check whether they have the same predicate symbol; this holds since we consider constant-free TGDs. Once we confirm that r_1 is compatible with r_2, we proceed with the construction of $\mathsf{R}(r_1, r_2)$. To this end, we need to construct $\theta = \mathsf{MGU}(head(r_1), body(r_2))$. Since θ consists of at most ω mappings $X \to Y$, where X and Y are variables occurring in Σ, it can be maintained in $O(\omega \log(\omega \cdot |\Sigma|))$ space. Now, to explicitly construct θ, we rely on a simplified version of the classical unification algorithm by Robinson.This algorithm constructs θ in an incremental way by scanning from left to right the variables of $head(r_1)$ and $body(r_2)$, and unify them whenever a disagreement is observed. Assuming that $head(r_1) = p(X_1, \ldots, X_n)$ and $body(r_2) = p(Y_1, \ldots, Y_n)$, the unification algorithm proceeds as follows:

1. $\theta := \big(\{X_i \to X_i\}_{i \in [n]} \cup \{Y_i \to Y_i\}_{i \in [n]}\big)$.
2. $ctr := 1$.
3. Repeat
   $V := X_{ctr}$ and $U := Y_{ctr}$.
   $V := \theta(V)$ and $U := \theta(U)$.
   If $V \neq U$, then $\theta := (\{U \to V\} \circ \theta)$.
   $ctr := ctr + 1$.
   Until $(ctr = n + 1)$.
4. Return θ.

The above algorithm runs in $O(\omega \log(\omega \cdot |\Sigma|))$ space, i.e., the space needed to maintain θ, V and U. Consequently, $\mathsf{R}(r_1, r_2)$ can be constructed using $O(\omega \log(\omega \cdot |sch(\Sigma)|) + \omega \log(\omega \cdot |\Sigma|))$ space. By providing similar analysis, we can show that the criticality check can be done using the same amount of space. Summing up, we get that the above nondeterministic procedure runs in $O(\omega \log(\omega \cdot |sch(\Sigma)|) + \omega \log(\omega \cdot |\Sigma|))$ space, as needed. This completes our proof. □

Proof of Lemma 4.40

It suffices to show that the complement of our problem is PSPACE-hard. The proof is by reduction from the acceptance problem of a deterministic polynomial space Turing machine M on an input $I = a_1 \ldots a_m$. Let $M = (S, \Lambda, \delta, s_0)$, where S is a finite set of states, $\Lambda = \{0, 1, \sqcup\}$ is the tape alphabet with $\sqcup$ be the blank symbol, $\delta : S \times \Lambda \to (S \times \Lambda \times \{\leftarrow, -, \rightarrow\})$ is the transition function, and $s_0 \in S$ is the initial state. We assume that M is well-behaved and never tries

to read beyond its tape boundaries, always halts, and uses exactly $n = m^k$ tape cells, where $k > 0$. We represent configurations using a string $\Lambda^\star S \Lambda^+$, i.e., the state of the configuration is placed to the immediate left of the cursor position; in this notation, the initial configuration is $s_0 a_1 \ldots a_m \sqcup^{n-m}$. Finally, we assume that in the accepting configuration the cursor points at the first tape cell.

Our goal is to construct a database D, a set $\Sigma \in (\mathsf{CT}^\star \cap \mathsf{L})$, where $\star \in \{\mathsf{obl}, \mathsf{sobl}\}$, and a propositional predicate *accept*, such that M accepts on input I iff $D \cup \Sigma \models \mathit{accept}$. In our construction we use the $(n+|S|+4)$-ary predicate *config* to represent the configurations of M. In the datbase D we store the initial configuration of M, while with Σ we simulate the transition function of M. Let us now formalize the above intuitive description. We assume the order $s_0 < s_1 < \ldots < s_{|S|-1}$ on the states of S, and we also assume that s_1 is the accepting state.

The database D consists of the single atom

$$\mathit{config}(s_0, a_1, \ldots, a_m, \underbrace{\sqcup, \ldots, \sqcup}_{n-m}, s_0, \ldots, s_{|S|-1}, 0, 1, \sqcup),$$

where the tuple $s_0, a_1, \ldots, a_m, \underbrace{\sqcup, \ldots, \sqcup}_{n-m}$ represents the initial configuration of M, the tuple $s_0, \ldots, s_{|S|-1}$ encodes the states of M, and the tuple $0, 1, \sqcup$ encodes the tape alphabet of M. The reason why we explicitly add the constants for the states and the tape symbols of M in the above atom, is to be able to access those constants without explicitly mention them in the TGDs.

The set Σ of linear TGDs is defined as follows:

- First, we simulate the transition function of M. We consider the three different cases where the cursor moves left, right, or stays at the same position.

  Left: For each transition rule $\delta(s_i, a) = (s_j, b, \leftarrow)$, we introduce, for each $\ell \in [n-1]$, the linear TGD:

$$\mathit{conf}(C_1, \ldots, C_{\ell-1}, T_i, A_a, C_{\ell+1}, \ldots, C_n, T_0, \ldots, T_{|S|-1}, A_0, A_1, A_\sqcup) \rightarrow$$
$$\mathit{conf}(C_1, \ldots, C_{\ell-2}, T_j, C_{\ell-1}, A_b, C_{\ell+1}, \ldots, C_n, T_0, \ldots, T_{|S|-1}, A_0, A_1, A_\sqcup).$$

  Right: For each transition rule of the form $\delta(s_i, a) = (s_j, b, \rightarrow)$, we introduce, for each $\ell \in [n-1]$, the linear TGD:

$$conf(C_1,\ldots,C_{\ell-1},T_i,A_a,C_{\ell+1},\ldots,C_n,T_0,\ldots,T_{|S|-1},A_0,A_1,A_\sqcup) \rightarrow conf(C_1,\ldots,C_{\ell-1},A_b,T_j,C_{\ell+1},\ldots,C_n,T_0,\ldots,T_{|S|-1},A_0,A_1,A_\sqcup).$$

Stay: For each transition rule of the form $\delta(s_i,a) = (s_j,b,-)$, we introduce, for each $\ell \in [n-1]$, the linear TGD:

$$conf(C_1,\ldots,C_{\ell-1},T_i,A_a,C_{\ell+1},\ldots,C_n,T_0,\ldots,T_{|S|-1},A_0,A_1,A_\sqcup) \rightarrow conf(C_1,\ldots,C_{\ell-1},T_j,A_b,C_{\ell+1},\ldots,C_n,T_0,\ldots,T_{|S|-1},A_0,A_1,A_\sqcup).$$

- Finally, once the accepting configuration is reached, the propositional atom *accept* is generated:

  $$conf(T_1,C_1,\ldots,C_n,T_0,\ldots,T_{|S|-1},A_0,A_1,A_\sqcup) \ \rightarrow \ accept.$$

  Recall that, by assumption, in the accepting configuration the cursor points at the first cell; thus, once the accepting configuration is reached, the above TGD will be triggered. In other words, we do not need to introduce TGDs where the body-variable T_1 appears in a position other than the first one.

Our construction is now complete. It is not difficult to show that M accepts on input I iff $D \cup \Sigma \models accept$. Since Σ does not contain existentially quantified variables, we immediately get that $\Sigma \in \mathsf{CT}^\star$, and the claim follows. □

Proof of Lemma 4.44

It suffices to show that $\star$-InfiniteDerivation runs in exponential space, in general, and in polynomial space, in the case of predicates of bounded arity; recall that AEXPSPACE = 2EXPTIME and APSPACE = EXPTIME. To this end, we show that the space required for the following tasks is exponential in the maximum arity ω of $sch(\Sigma)$, and polynomial in all the other parameters of the input: (1) maintain the cloud of an atom; (2) maintain the set H_r, where $r \in \Sigma$; (3) maintain the integer value of ctr; and (4) verify that the guessed cloud is valid. Let us analyze the above tasks:

1. In the cloud of an atom A, by definition, we have only predicates of $sch(\Sigma)$ and terms of $(dom(A) \cup \{\mathsf{c}\})$. Since $|(dom(A) \cup \{\mathsf{c}\})| \leqslant (\omega+1)$, we conclude that $|cloud(A,D,\Sigma)| \leqslant |sch(\Sigma)| \cdot (\omega+1)^\omega$, and thus it can be maintained in exponential space in ω.

2. H_r may contain all the possible homomorphisms from $var(guard(r))$ to $dom(A)$, where A is the last atom generated by the algorithm. Since $|var(guard(r))| \leqslant \omega$ and $|dom(A)| \leqslant \omega$, we get that $|H_r| \leqslant \omega^{\omega}$, and thus it can be maintained in exponential space in ω.
3. Recall that δ is double-exponential in ω, and only ω appears in the second exponent. Hence, the integer value of ctr, which is at most $(2 \cdot \delta)$, can be represented in binary form using space $O(\log \delta)$.
4. Finally, it is known that the task of verifying whether the guessed cloud is valid, can be carried out by an alternating procedure that uses exponential space in ω. The rather involved algorithm is thoroughly explained in [22], where the main problem is query answering.

This completes our proof. □

Proof of Lemma 4.45

Here we focus on the 2EXPTIME-hardness. The EXPTIME-hardness can be established in a similar way. For technical reasons, we focus on *standard databases*, that is, databases that have at least two constants, let say 0 and 1, that are available via the unary predicates $0(\cdot)$ and $1(\cdot)$, respectively. Our proof is obtained by a significant modification of the second part of the proof of Theorem 6.2 in [22], which shows the 2EXPTIME-hardness of checking whether $D \cup \Sigma \models q$, where D is a database, Σ is a set of guarded TGDs, and q is a propositional predicate. There, this is shown using a fixed database consisting of the single atom $zeroone(0, 1)$ that, in essence, just defines the distinct 0 and 1 constants. We use the same database D in the present proof, with the additional assumption that D is standard. The presentation here is self-contained.

Given that AEXPSPACE = 2EXPTIME (AEXPSPACE stands for alternating EXPSPACE), our aim is, as in the proof of Theorem 6.2 of [22], to simulate an AEXPSPACE Turing machine. As already noted there, it is sufficient to simulate one that uses no more than 2^n worktape cells, since the acceptance problem for such machines is already 2EXPTIME-hard. In fact, by trivial padding arguments, the acceptance problem for every AEXPSPACE machine can be transformed in polynomial time into the acceptance problem for an alternating linear space (ALINSPACE) machine using at most 2^n worktape cells. We will thus concentrate on such machines. In the present proof, we make, without loss of generality , an additional assumption: we assume the machine contains a counter of 2^{n-1} bits (i.e., the second half of the worktape) that is initialized to

zero and can count from 0 up to $2^{2^{n-1}}-1$, and which is regularly incremented by adding 1 after $O(2^n)$ steps until either the machine stops or the counter reaches the maximal value of $2^{2^{n-1}}-1$ (which consists of 2^{n-1} ones), in which case the machine is forced to stop in a rejecting state. This makes sure that the machine cannot cycle and always stops within $O(2^{2^n})$ steps. Adding such counters to a Turing machine, giving rise to the concept of *clocked Turing Machine*, is a well-known technique[2]; see also [59, 74]. Given that counting is in LOGSPACE, every ALINSPACE Turing machine with workspace N can be augmented with a counter with mild overhead only. The problem of deciding whether an AEXPSPACE machine M equipped with such a counter accepts its input I is thus still 2EXPTIME-complete. We finally assume, again without loss of generality, that the counter always occupies the last (i.e., rightmost) 2^{n-1} bits of M's worktape. Finally, we assume, again without loss of generality, that the alphabet is $\{0,1\}$, and that in the initial configuration, the machine stores the input string I in the leftmost part of the worktape, and that all cells to the right of it (including, of course, the counter) contain the value zero.

To simulate the computation of an ALINSPACE Turing machine M as above on input I, in the proof of Theorem 6.2 of [22], an infinite binary alternation tree is generated where the two children Y and Z of a node X are the identifiers of the two successor configurations of configuration X. However, this tree does not need to be infinite. Its branches can be cut off at a suitable double-exponential depth, because beyond that depth some earlier configuration of the same branch must have been repeated, and it is thus useless to pursue the branch further. We will thus modify the proof of Theorem 6.2 in [22], so that a node generates children, only if the node does not exceed a certain depth. For technical reasons, in our present proof, just as in the proof of Theorem 6.2 in [22], the first two arguments of some atoms below will be dummy variables T_0 and T_1 that will always be forced to take the values 0 and 1, respectively. This way, where convenient, we will have the values 0 and 1 available implicitly in form of variables, and we will not need to use these constants explicitly in our TGDs.

Our simulation starts by selecting the constant 0 as the identifier for the initial configuration, which is done by the TGD

$$0(T_0), 1(T_1), \mathit{zeroone}(T_0,T_1) \;\rightarrow\; \mathit{init}(T_0,T_1,T_0).$$

[2] http://cfcul.fc.ul.pt/publicacoes/artigos/BSPM_Felix_N67.pdf

The first two arguments of $\mathit{init}(T_0, T_1, T_0)$ just serve, as explained, to carry the values 0 and 1 along. We also add the TGD

$$0(T_0), 1(T_1), \mathit{init}(T_0, T_1, T_0) \;\rightarrow\; \mathit{config}(T_0, T_1, T_0)$$

to assert that 0 is indeed a configuration. Further below we will continue to define the initial configuration by generating exponentially many tape cells, associating them with the identifier "0" of the initial configuration, and initializing their value with the input string I followed by zeroes. Let us, however, first describe how new configurations are generated from the initial one. In the proof of Theorem 6.2 in [22], the configuration tree generation rules were as follows:

$$\begin{aligned} 0(T_0), 1(T_1), \mathit{config}(T_0, T_1, X) &\rightarrow \exists Y \exists Z\, \mathit{next}(T_0, T_1, X, Y, Z), \\ 0(T_0), 1(T_1), \mathit{next}(T_0, T_1, X, Y, Z) &\rightarrow \mathit{config}(T_0, T_1, Y), \\ 0(T_0), 1(T_1), \mathit{next}(T_0, T_1, X, Y, Z) &\rightarrow \mathit{config}(T_0, T_1, Z). \end{aligned}$$

The first TGD generates to each configuration identifier X two children Y and Z, which are meant to be the identifiers of X's successor configurations. The other two TGDs just say that each successor configuration is a configuration. Obviously these rules enforce a non-terminating (infinite) Chase, and they cannot be used for present purposes. However, to ensure that the Chase terminates, the following minor modification of the first rule, while keeping the other two TGDs, will suffice:

$$0(T_0), 1(T_1), \mathit{config}(T_0, T_1, X), \mathit{valid}(X) \;\rightarrow\; \exists Y \exists Z\, \mathit{next}(T_0, T_1, X, Y, Z).$$

Here, the atom $\mathit{valid}(X)$ will ensure that only "valid" configurations give rise to children, which are those whose counter holds a value smaller than $2^{2^{n-1}}-1$. In practice, this means that at least one of the last 2^{n-1} bits must be zero. To define the $\mathit{valid}(\cdot)$ predicate via guarded TGDs, we thus just need to check the latter property. We will do that after we have explained how to encode the worktape associated with each configuration X.

Towards a suitable representation of the worktape cells, and to access these cells and put them in relation, we use a number of guarded TGDs to create a predicate b whose extension shall contain all atoms

$$b(0, 1, \mathbf{v}, \mathbf{u}, x, y, z)$$

such that $\mathbf{v}$ is a vector of n bits, $\mathbf{u}$ is an n-bit vector *different from* $\mathbf{v}$, and x, y, z are configuration identifiers, as above. For better readability, whenever useful, we will use superscripts for indicating the arity of vector variables: for instance, $\mathbf{V}^{(r)}$ denotes $V_1, \ldots, V_r$. Boldface variable symbols without superscript always indicate n-ary vectors. Instead, a repeated variable or constant will have the total number of repetitions as superscript, thus T_0^3 stands for (T_0, T_0, T_0). To define the predicate b using guarded TGDs, we first define a predicate $b'(0, 1, \mathbf{v}, \mathbf{u}, x, y, z)$ which is as b, except that $\mathbf{v}$ and $\mathbf{u}$ may also be equal, and then getting b from b'. This goes as follows. We start with the TGD

$$0(T_0), 1(T_1), \mathit{next}(T_0, T_1, X, Y, Z) \;\rightarrow\; b'(T_0, T_1, T_0^n, T_0^n, X, Y, Z),$$

which defines an atom $b'(0, 1, 0^n, 0^n, x, y, z)$, for each configuration x with its *next*-successors y and z. The following $2n$ TGDs, for $1 \leqslant i \leqslant n$, generate an exponential number of new atoms, for each triple X, Y, Z, by swapping 0s to 1s in all possible ways in the bit vectors $\mathbf{v}$ and $\mathbf{u}$ of b'. Eventually, we obtain all combinations of possible values for $\mathbf{v}$ and $\mathbf{u}$.

$$0(T_0), 1(T_1), b'(T_0, T_1, V_1, \ldots, V_{i-1}, T_0, V_{i+1}, \ldots, V_n, \mathbf{U}, X, Y, Z) \rightarrow \\ b'(T_0, T_1, V_1, \ldots, V_{i-1}, T_1, V_{i+1}, \ldots, V_n, \mathbf{U}, X, Y, Z)$$

$$0(T_0), 1(T_1), b'(T_0, T_1, \mathbf{V}, U_1, \ldots, U_{i-1}, T_0, U_{i+1}, \ldots, U_n, X, Y, Z) \rightarrow \\ b'(T_0, T_1, \mathbf{V}, U_1, \ldots, U_{i-1}, T_1, U_{i+1}, \ldots, U_n, X, Y, Z).$$

It is trivial to define using guarded full TGDs a vectorized $2n$-ary *neq* relation such that $\mathit{neq}(V^n, U^n)$ is true iff V^n and U^n are different bit vectors. Using this, we define b from b' as follows:

$$0(T_0), 1(T_1), b'(T_0, T_1, \mathbf{V}, \mathbf{U}, X, Y, Z), \mathit{neq}(\mathbf{V}, \mathbf{U}) \;\rightarrow\; b(T_0, T_1, \mathbf{V}, \mathbf{U}, X, Y, Z).$$

In order to simulate the Turing machine M, we will represent each of the exponentially many worktape cells by an n-ary bit vector. For example, the fact that at configuration c the head of the machine is over cell $\mathbf{i}$ will be expressed by a predicate $\mathit{head}(\mathbf{i}, c)$, where $\mathbf{i}$ is an n-ary bit-vector and c a configuration identifier. We then need to use TGDs to express transitions such as: $(s, 0 \rightarrow 1Rs'; 0Ls'')$ meaning that if the machine M in some configuration c is in state s and contains 0 in the current cell (say cell $\mathbf{i}$), then obtain the first

successor configuration c_1 of c by writing 1 into cell $\mathbf{i}$ and move the cursor to the right (over cell $\mathbf{i}+1$) and switch to state s', and obtain the second successor configuration c_2 of c by keeping 0 in cell $\mathbf{i}$, moving to the left and switching to state s''. To express such transitions with guarded rules, we need a very powerful guard relation g that at the same time can speak about bit-vector successors and configuration-successors. The guard-relation g is defined from b through the following group of guarded rules: For each $0 \leqslant r < n$, we add:

$$0(T_0), 1(T_1), b(T_0, T_1, \mathbf{V}^{(r)}, T_0, T_1^{n-r-1}, \mathbf{U}, X, Y, Z) \ \rightarrow \\ g(T_0, T_1, \mathbf{V}^{(r)}, T_0, T_1^{n-r-1}, \mathbf{V}^{(r)}, T_1, T_0^{n-r-1}, \mathbf{U}, X, Y, Z).$$

The above n rules define an exponential number of cell-successor pairs for each triple of configuration identifiers X, Y, Z, where Y and Z are the "next" configurations following X. In particular, the relation g contains precisely all tuples $g(0, 1, \mathbf{v}, \mathbf{w}, \mathbf{u}, x, y, z)$, such that $\mathbf{v}$ is an n-ary bit vector, $\mathbf{w}$ is its binary successor, $\mathbf{u}$ is an n-ary bit-vector different from $\mathbf{v}$, x is a configuration identifier, y is its first successor via the *next* relation, and z is its second successor via the *next* relation.

We are now ready to simulate an AEXPSPACE Turing machine M over an input string I by a set Σ of guarded TGDs. This simulation is similar to the one presented in the proof of Theorem 6.2 of [22]. We first explain the used predicates:

- *state*(, X): the state of configuration X is . Here stands for a bit-vector of length $\lceil \log |S| \rceil$, where S is the state-set of machine M. When appears in a TGD, this is just an abbreviation for the explicit binary representation of state s. For example, if s is the fifth state in a fixed order, then $= (101)$, or, if one prefers not to use the constants directly, $= (T_1, T_0, T_1)$. (Remember that T_0 and T_1 are always bound to 0 and 1, respectively.) We assume, w.l.o.g., that there is a unique accepting state and denote it by $\mathbf{s_a}$.
- *cell*($\mathbf{V}$): the extension of this predicate contains all tape cell indexes.
- *head*($\mathbf{V}, X$): at configuration X the workhead is over cell $\mathbf{V}$. Here $\mathbf{V}$ is a n-ary list (vector) of variables.
- *zero*($\mathbf{V}, X$) and *one*($\mathbf{V}, X$): the content of worktape cell $\mathbf{V}$ of configuration X is zero/o ne.
- *accepting*(X): X is an accepting configuration.
- *existential*(X) and *universal*(X): these are predicates that specify whether a configuration X is existential or universal.

- *Comparison predicates on bit vectors*: We assume that, in addition to the already mentioned $2n$-ary predicate *neq*, we have all usual comparison predicates on n-ary bit vectors at our disposal, in particular $\leq$ and $<$. It is easy to see that such predicates can be implemented via a polynomially sized set of full guarded TGDs.

We add to Σ the TGD

$$\mathit{state}(X, \mathbf{s_a}) \;\rightarrow\; \mathit{accepting}(X),$$

which just says that a configuration is accepting if its state is the accepting state. We also add the following TGD defining the above-mentioned n-ary predicate *cell*:

$$0(T_0), 1(T_1), b(T_0, T_1, \mathbf{V}, \mathbf{U}, X, Y, Z) \rightarrow \mathit{cell}(\mathbf{V}).$$

Before describing the simulation of M's transition function, we show how the worktape content is initialized to the input value I. For each $0 \leqslant i < 2^n$, if the i-th bit of I is zero, then we assert the TGD

$$0(T_0), 1(T_1), \mathit{zeroone}(T_0, T_1) \rightarrow \mathit{zero}([\mathbf{j}], T_0),$$

where $[\mathbf{j}]$ denotes an appropriate sequence of variables from $\{T_0, T_1\}$, which in turn encodes the binary number of length n corresponding to $i - 1$ (the tape starts at position 0). Similarly, we assert

$$0(T_0), 1(T_1), \mathit{zeroone}(T_0, T_1) \rightarrow \mathit{one}([\mathbf{j}], T_0),$$

in case the i-th bit of I is one. If k (in binary $\mathbf{k}$) is the index of the rightmost tape cell of I, then all cells to the right of it should be filled with zeroes. This is done by the following TGD:

$$0(T_0), 1(T_1), b(T_0, T_1, \mathbf{V}, \mathbf{U}, X, Y, Z), < ([\mathbf{k}], \mathbf{V}) \rightarrow \mathit{zero}(\mathbf{V}, T_0),$$

where $[\mathbf{k}]$ is the sequence of variables from $\{T_0, T_1\}$ corresponding to $\mathbf{k}$. (We will use this notation in what follows without further explanation.) We initialize the workhead position for configuration 0 to be on cell with index 0^n:

$$0(T_0), 1(T_1), \mathit{zeroone}(T_0, T_1) \;\rightarrow\; \mathit{head}(T_0^n, T_0).$$

Moreover, for the initial state s_0 with binary representation $\mathbf{s_0}$, we assert:

$$0(T_0), 1(T_1), zeroone(T_0, T_1) \;\rightarrow\; state([\mathbf{s_0}], T_0).$$

A configuration is an existential one iff its state is existential; otherwise it is universal. Thus, for each existential state s (whose binary encoding is $\mathbf{s}$) we add the TGD:

$$0(T_0), 1(T_1), b(T_0, T_1, \mathbf{V}, \mathbf{U}, X, Y, Z), state([\mathbf{s}], X) \rightarrow existential(X),$$

and for each universal state s we add the TGD:

$$0(T_0), 1(T_1), b(T_0, T_1, \mathbf{V}, \mathbf{U}, X, Y, Z), state([\mathbf{s}], X) \rightarrow universal(X).$$

We are now finally ready to describe how the transitions are translated into guarded rules. We exemplify that on hand of the above described transition $(s, 0 \rightarrow 1Rs'; 0Ls'')$. For this transition, we assert the following TGDs:

$$\begin{aligned} 0(T_0), 1(T_1), g(T_0, T_1, \mathbf{V}, \mathbf{W}, \mathbf{U}, X, Y, Z), state([\mathbf{s}], X), head(\mathbf{V}, X), zero(\mathbf{V}, X) \\ \rightarrow\; one(\mathbf{V}, Y), state([\mathbf{s}'], Y), head(\mathbf{W}, Y) \end{aligned}$$

$$\begin{aligned} 0(T_0), 1(T_1), g(T_0, T_1, \mathbf{V}, \mathbf{W}, \mathbf{U}, X, Y, Z), state([\mathbf{s}], X), head(\mathbf{W}, X), zero(\mathbf{W}, X) \\ \rightarrow\; zero(\mathbf{W}, Z), state([\mathbf{s}''], Z), head(\mathbf{V}, Z). \end{aligned}$$

We also need to add the so-called inertia rules, which state that all cells that are not under the cursor keep their content in the subsequent configurations:

$$\begin{aligned} 0(T_0), 1(T_1), b(T_0, T_1, \mathbf{V}, \mathbf{U}, X, Y, Z), head(\mathbf{V}, X), zero(\mathbf{U}, X) \;\rightarrow \\ zero(\mathbf{U}, Y), zero(\mathbf{U}, Z) \end{aligned}$$

$$\begin{aligned} 0(T_0), 1(T_1), b(T_0, T_1, \mathbf{V}, \mathbf{U}, X, Y, Z), head(\mathbf{V}, X), one(\mathbf{U}, X) \;\rightarrow \\ one(\mathbf{U}, Y), one(\mathbf{U}, Z). \end{aligned}$$

These rules altogether precisely simulate transition $(s, 0 \rightarrow 1Rs'; 0Ls'')$, and all other transitions can be simulated by similar TGDs. Let us now show how the *valid* predicate is implemented:

$$0(T_0), 1(T_1), b(T_0, T_1, \mathbf{V}, \mathbf{U}, X, Y, Z), < (T_0, T_1^{n-1}, \mathbf{V}), zero(\mathbf{V}, X) \rightarrow valid(X).$$

This TGD just states that if in state X there exists a cell among the last 2^{n-1} cells whose content is zero, then the counter is not at its maximum value, and the configuration is valid. This makes sure that the Chase always terminates: given that the counter increases monotonically and is periodically incremented after $O(2^n)$ steps, and given that on a branch of the alternation tree the machine never runs into the same configuration, on each corresponding branch of the guarded Chase forest, only $O(2^{2^n})$ nulls are created, and the Chase eventually stops. What remains to be defined are the *acceptance rules* that derive the predicate *accept* if M accepts I. Basically, this is achieved by working the alternating configuration tree upwards and propagating an *accepting* predicate from the leaves to the root, i.e., to the configuration with identifier 0:

$$existential(X), next(X, X_1, X_2), accepting(X_1) \rightarrow accepting(X)$$
$$existential(X), next(X, X_1, X_2), accepting(X_2) \rightarrow accepting(X)$$
$$universal(X), next(X, X_1, X_2), accepting(X_1), accepting(X_2) \rightarrow accepting(X)$$
$$0(T_0), 1(T_1), zeroone(T_0, T_1), accepting(T_0) \rightarrow accept.$$

This completes the description of Σ. Note that $\Sigma \in (\mathsf{CT}^{\star} \cap \mathsf{G})$, where $\star \in \{\mathsf{obl}, \mathsf{sobl}\}$, if we focus on standard databases. It is also interesting to observe that $|sch(\Sigma)|$ is bounded by an integer constant. Moreover, Σ can be obtained in LOGSPACE from I and the machine description of M, and faithfully simulates the behavior of the alternating exponential space machine M on input I via a terminating Chase. Therefore, $D \cup \Sigma \models accept$ iff M accepts input I, and the claim follows.

Remark 1: In case of non-standard databases, i.e., databases with only one constant, the problem is in 2EXPTIME (this follows from Theorem 4.41) and EXPTIME-hard. The lower bound is obtained by first showing that propositional atom entailment under guarded Datalog programs is EXPTIME-hard, and then applying Proposition 4.36.

Remark 2: Regarding the 2EXPTIME (resp., EXPTIME) upper bound in the case of standard databases, in fact we should start from a database that contains a bounded number of constants, and not from the critical database since the critical database is not a standard one. More precisely, given a set

$\Sigma \in \mathsf{WG}$, we can show the following: for every standard database D, the $\star$-chase of D w.r.t. Σ terminates iff for every standard database D' such that $|dom(D')| \leqslant 2 \cdot b \cdot \omega + 2$, the $\star$-chase of D' w.r.t. Σ terminates, where b is the maximum number of body-atoms over all TGDs of Σ. Thus, during the execution of $\star$-$\mathsf{InfiniteDerivation}$, we first need to guess a standard database D with the above property, and the rest of the algorithm is exactly the same. Notice that D is of exponential size, in general, and of polynomial size in case of bounded arity.

Proofs of Section 4.4

Before presenting our proofs, we introduce some notions used hereafter.

Let $h : dom(A_1) \rightarrow dom(A_2)$ be a homomorphism from a set of atoms A_1 to a set of atoms A_2. The restriction of h to a subset X of $dom(A_1)$ is denoted as $h|_X$.

Let S be a (possibly infinite) sequence of Chase steps $K_0 \stackrel{r_0,h_0}{\rightarrow} K_1 \stackrel{r_1,h_1}{\rightarrow} ...$. We use S^i to denote the sub-sequence consisting of the first i Chase steps of S. Moreover, given $t_1, t_2 \in const \cup null$, we write $t_1 \approx_S t_2$ iff

- *(base case)* $t_1 = t_2$ or there is a Chase step $K_i \stackrel{r_i,h_i}{\rightarrow} K_{i+1}$ in S s.t. r_i is an EGD $\varphi(\mathbf{X}, \mathbf{Y}) \rightarrow X_1 = X_2$, $t_1, t_2 \in \{h_i(X_1), h_i(X_2)\}$, and $K_{i+1} \neq \perp$;
- *(iterative case)* there exists $t_3 \in const \cup null$ such that $t_1 \approx_S t_3$ and $t_3 \approx_S t_2$.

Intuitively, $t_1 \approx_S t_2$ means that t_1 and t_2 are equal or have been made equal (by some EGD) in the sequence S.

We extend the $\approx_S$ notation to homomorphisms as follows: given two homomorphisms $h_i : T \rightarrow T_i$ and $h_j : T \rightarrow T_j$, we write $h_i \approx_S h_j$ iff $h_i(t) \approx_S h_j(t)$ for every $t \in T$.

Please note that, in the light of the definitions above, the notions of oblivious (resp. semi-oblivious) Chase sequence can be equivalently defined as follows. An *oblivious* (resp. *semi-oblivious*) *Chase sequence S of D with Σ* is an exhaustive application of Chase steps $K_0 \stackrel{r_0,h_0}{\rightarrow} K_1 \stackrel{r_1,h_1}{\rightarrow} ...$ such that

- $K_0 = D$,
- every $r_i \in \Sigma$, and
- for every Chase step $K_i \stackrel{r_i,h_i}{\rightarrow} K_{i+1}$, there is no Chase step $K_j \stackrel{r_j,h_j}{\rightarrow} K_{j+1}$ such that $j < i$, $r_i = r_j = r$, and $h_j \approx_{S^i} h_i$ (resp. $h_j|_{fr(r)} \approx_{S^i} h_i|_{fr(r)}$).

Roughly speaking, the last item above says that in order for $K_i \stackrel{r,h_i}{\to} K_{i+1}$ to be a Chase step in an oblivious (resp. semi-oblivious) Chase sequence, another Chase step $K_j \stackrel{r,h_j}{\to} K_{j+1}$ with $h_j \approx_{S^i} h_i$ (resp. $h_j|_{fr(r)} \approx_{S^i} h_i|_{fr(r)}$) must not have been applied before in the sequence—notice that the definitions of oblivious and semi-oblivious Chase sequence differ only in how h_i and h_j are compared.

Lemma 6.2. *Let D be a database and Σ a set of dependencies.*

1. *If S is a standard (resp. semi-oblivious) Chase sequence of D with Σ, then S is a semi-oblivious (resp. oblivious) Chase sequence of D with Σ.*
2. *If S is a terminating oblivious (resp. semi-oblivious) Chase sequence of D with Σ, then there is a subsequence of S which is a terminating semi-oblivious (resp. standard) Chase sequence of D with Σ.*

Proof. 1. Let S be a standard Chase sequence of D with Σ. By contradiction, assume that S is not a semi-oblivious Chase sequence of D with Σ. Then there is a pair of Chase steps $K_i \stackrel{r_i,h_i,\gamma_i}{\longrightarrow} K_{i+1}$ and $K_j \stackrel{r_j,h_j,\gamma_j}{\longrightarrow} K_{j+1}$ of S such that *i)* $j < i$, *ii)* $r_j = r_i = r$ and *iii)* $h_j|_{fr(r)} \approx_{S^i} h_i|_{fr(r)}$. Notice that this means that K_j satisfies the dependency obtained by applying h_j to r_j and thus $K_j \stackrel{r_j,h_j,\gamma_j}{\longrightarrow} K_{j+1}$ is not a standard Chase step, which is a contradiction. An analogous reasoning can be applied to show that if S is a semi-oblivious Chase sequence of D with Σ, then it is also an oblivious one.

2. Let S be a terminating oblivious Chase sequence of D with Σ. By iteratively deleting from S each Chase step $K_i \stackrel{r_i,h_i,\gamma_i}{\longrightarrow} K_{i+1}$ such that there is another Chase step $K_j \stackrel{r_j,h_j,\gamma_j}{\longrightarrow} K_{j+1}$ in S, with $j < i$, $r_i = r_j = r$, and $h_j|_{fr(r)} \approx_{S^i} h_i|_{fr(r)}$, we obtain a terminating semi-oblivious Chase sequence of D with Σ. An analogous reasoning can be applied to show that if S is a terminating semi-oblivious Chase sequence of D with Σ, then it is also a terminating standard Chase sequence of D with Σ. □

Proof of Theorem 4.47

We start with the following two simple properties. Let D be a database and Σ a set of dependencies.

Property 1. If S is a standard (resp. semi-oblivious) Chase sequence of D with Σ, then S is a semi-oblivious (resp. oblivious) Chase sequence of D with Σ.

Property 2. If S is a terminating oblivious (resp. semi-oblivious) Chase sequence of D with Σ, then there is a subsequence of S which is a terminating semi-oblivious (resp. standard) Chase sequence of D with Σ.

1. Obviously, $\mathsf{CT}_\forall^\star \subseteq \mathsf{CT}_\exists^\star$ for every $\star \in \{\mathsf{std}, \mathsf{obl}, \mathsf{sobl}, \mathsf{core}\}$. To prove the proper containment for $\star \in \{\mathsf{std}, \mathsf{obl}, \mathsf{sobl}\}$, we show a set of dependencies Σ such that $\Sigma \in \mathsf{CT}_\exists^\star$ and $\Sigma \notin \mathsf{CT}_\forall^\star$. To this end, consider the set Σ consisting of the following two dependencies:

$$e(X,Y) \rightarrow \exists Z\ e(Y,Z)$$
$$e(X,Y) \wedge e(Y,Z) \rightarrow X = Z$$

   For $\star \in \{\mathsf{obl}, \mathsf{sobl}, \mathsf{std}\}$, $\Sigma \notin \mathsf{CT}_\forall^\star$, as starting from the database $D = \{e(a,b)\}$, it is possible to derive a non-terminating $\star$-chase sequence of D with Σ by repeatedly applying the TGD in Σ. On the other hand, for every database D, there exists a terminating $\star$-chase sequence of D with Σ, where each Chase step using the TGD is followed by a Chase step using the EGD.
   To show that $\mathsf{CT}_\forall^{\mathsf{core}} = \mathsf{CT}_\exists^{\mathsf{core}}$, recall that for every database D and set Σ of dependencies, all core Chase sequences of D with Σ are isomorphically equivalent. This implies that all core Chase sequences have the same number of core Chase steps and thus, if $\Sigma \in \mathsf{CT}_\exists^{\mathsf{core}}$, then $\Sigma \in \mathsf{CT}_\forall^{\mathsf{core}}$.
2. The first item of Lemma 6.2 implies that $\mathsf{CT}_\forall^{\mathsf{obl}} \subseteq \mathsf{CT}_\forall^{\mathsf{sobl}} \subseteq \mathsf{CT}_\forall^{\mathsf{std}}$. The second item of Lemma 6.2 implies that $\mathsf{CT}_\exists^{\mathsf{obl}} \subseteq \mathsf{CT}_\exists^{\mathsf{sobl}} \subseteq \mathsf{CT}_\exists^{\mathsf{std}}$.
   The relation $\mathsf{CT}_\mathsf{q}^{\mathsf{std}} \subseteq \mathsf{CT}_\mathsf{q}^{\mathsf{core}}$, for $\mathsf{q} \in \{\exists, \forall\}$, can be proved as follows. If every (or at least one) standard Chase sequence is terminating, then it gives a universal model [38]. As the core Chase is a complete procedure for computing universal models [33], then there must exist a terminating core Chase sequence constructing a universal model as well. This shows $\mathsf{CT}_\mathsf{q}^{\mathsf{std}} \subseteq \mathsf{CT}_\exists^{\mathsf{core}}$, for $\mathsf{q} \in \{\exists, \forall\}$. Since $\mathsf{CT}_\forall^{\mathsf{core}} = \mathsf{CT}_\exists^{\mathsf{core}}$ (as shown in point 1 of this proof), then $\mathsf{CT}_\mathsf{q}^{\mathsf{std}} \subseteq \mathsf{CT}_\mathsf{q}^{\mathsf{core}}$, for $\mathsf{q} \in \{\exists, \forall\}$.
   All proper containments follow from the fact that for sets of TGDs only, the following relations hold: $\mathsf{CT}_\forall^{\mathsf{obl}} = \mathsf{CT}_\exists^{\mathsf{obl}} \subsetneq \mathsf{CT}_\forall^{\mathsf{sobl}} = \mathsf{CT}_\exists^{\mathsf{sobl}} \subsetneq \mathsf{CT}_\forall^{\mathsf{std}} \subsetneq \mathsf{CT}_\exists^{\mathsf{std}} \subsetneq \mathsf{CT}_\forall^{\mathsf{core}} = \mathsf{CT}_\exists^{\mathsf{core}}$.
3. As recalled in Section 4.1, for sets of TGDs, the following proper inclusions hold: $\mathsf{CT}_\exists^{\mathsf{obl}} \subsetneq \mathsf{CT}_\forall^{\mathsf{sobl}}$, $\mathsf{CT}_\exists^{\mathsf{sobl}} \subsetneq \mathsf{CT}_\forall^{\mathsf{std}}$, and $\mathsf{CT}_\exists^{\mathsf{obl}} \subsetneq \mathsf{CT}_\forall^{\mathsf{std}}$. Therefore, it suffices to show that for dependencies containing also EGDs the following relations hold: $\mathsf{CT}_\exists^{\mathsf{obl}} \not\subseteq \mathsf{CT}_\forall^{\mathsf{sobl}}$, $\mathsf{CT}_\exists^{\mathsf{sobl}} \not\subseteq \mathsf{CT}_\forall^{\mathsf{std}}$, and $\mathsf{CT}_\exists^{\mathsf{obl}} \not\subseteq \mathsf{CT}_\forall^{\mathsf{std}}$. To

prove these relations, it suffices to consider the set of dependencies Σ reported in point 1) of this proof: as already discussed, $\Sigma \in \mathsf{CT}^{\star}_{\exists}$ for $\star \in \{\mathsf{obl}, \mathsf{sobl}, \mathsf{std}\}$, and $\Sigma \notin \mathsf{CT}^{\star}_{\forall}$ for $\star \in \{\mathsf{obl}, \mathsf{sobl}, \mathsf{std}\}$. $\square$

Proof of Theorem 4.49

1. Let $\Sigma' = \Sigma_{ax} \cup \Sigma_{sim}$, where Σ_{ax} is the set of the (full) TGDs defining the equality-axioms of the substitution-free simulation of Σ, whereas Σ_{sim} is the remaining set of TGDs of Σ'. We recall that Σ_{ax} consists of the following (full) TGDs:
   - $eq(X,Y) \rightarrow eq(Y,X)$;
   - $eq(X,Y) \wedge Eq(Y,Z) \rightarrow eq(X,Z)$;
   - For every $p \in \mathcal{R}$ with $arity(p) = n$,
     $p(X_1,...,X_n) \rightarrow Eq(X_1,X_1) \wedge ... \wedge eq(X_n,X_n)$

   while Σ_{sim} is obtained from Σ as follows:
   - every equality $X = Y$ in the head of an EGD is replaced with $eq(X,Y)$;
   - $n \geq 2$ occurrences of the same variable X in the body of a dependency are replaced with $X, X_1, X_2, ..., X_{n-1}$ and the conjunction $eq(X,X_1) \wedge eq(X_1,X_2) \wedge ... \wedge eq(X_{n-2},X_{n-1})$ is added to the body, where the X_i's are fresh variables;
   - all occurrences of a constant c in the body of a dependency are replaced by a fresh variable X_c and the atom $eq(X_c,c)$ is added to the body conjunction.

   Given a dependency $r \in \Sigma$, we denote by r' the dependency in Σ_{sim} obtained from r by applying the aforementioned rewriting.
   Note that every dependency in Σ_{ax} is a full TGD. Thus, given an instance K, there is no infinite $\star$-chase sequence of K with Σ_{ax} and all terminating $\star$-chase sequences of K with Σ_{ax} yield the same result instance. In the following, we denote by $K\,chase^{\Sigma_{ax}}(J,)$ any of the terminating $\star$-chase sequences of K with Σ_{ax} having J as result; obviously, $K \subseteq J$.
   We now prove that $\Sigma \notin \mathsf{CT}^{\star}_{\forall}$ implies $\Sigma' \notin \mathsf{CT}^{\star}_{\forall}$. Let D be a database such that there is an infinite $\star$-chase sequence S of D with Σ. We can build an infinite $\star$-chase sequence S' of D with Σ' from S by *(i)* enforcing Σ_{ax} at the beginning and after each Chase step of S, and then *(ii)* deleting Chase steps that violate the definition of $\star$-chase sequence. Specifically, S' is obtained from S as follows.

(i) Each Chase step $K_i \xrightarrow{r_i,h_i,\gamma_i} K_{i+1}$ (with $i = 0, 1, ...$) in S is replaced with the Chase sequence $K'_i chase^{\Sigma_{ax}}(J_,)\, i \xrightarrow{r'_i,h'_i,\gamma'_i} K'_{i+1}$, where

a) $K'_0 = D$,
b) K'_{i+1} is defined according to the definition of Chase step,
c) h'_i is a homomorphism from $body(r'_i)$ to J_i s.t. $h'_i|_{dom(body(r_i))} \approx_{S^i} h_i$,
d) γ'_i is the empty substitution.

Since $h_i(body(r_i)) \subseteq K_i$, $K'_i \subseteq J_i$, and all replacements made by an EGD in S are derived as eq-atoms by enforcing Σ_{ax}, then it can be easily shown by induction on i that the homomorphisms h'_i above always exist.

(ii) Then, every Chase step in S' violating the condition imposed by the definition of $\star$-chase sequence is deleted from S'. Indeed, deletions can only occur in sequences of the form $K'_i chase^{\Sigma_{ax}}(J_,)\, i$ and thus S' is infinite. To show that none of the Chase steps of S' using TGDs of Σ_{sim} is deleted, we first consider the case where $\star = \mathsf{obl}$ (resp. $\star = \mathsf{sobl}$). Note that for every pair of Chase steps $K_i \xrightarrow{r_i,h_i,\gamma_i} K_{i+1}$, $K_j \xrightarrow{r_j,h_j,\gamma_j} K_{j+1}$ in S such that $i < j$ and $r_i = r_j = r$, then $h_i \not\approx_{S^j} h_j$ (resp., $h_i|_{fr(r)} \not\approx_{S^j} h_j|_{fr(r)}$). Since no EGD occur in Σ', then no distinct terms are identified in S', and thus it is the case that for every pair of Chase steps $J_i \xrightarrow{r'_i,h'_i,\gamma'_i} K'_{i+1}$, $J_j \xrightarrow{r'_j,h'_j,\gamma'_j} K'_{j+1}$, with $i < j$, if $r'_i = r'_j = r'$, then $h'_i \neq h'_j$ (resp. $h'_i|_{fr(r')} \neq h'_j|_{fr(r')}$). For $\star = \mathsf{std}$, it can be easily seen by induction on i that if there is no extension g_i of h_i such that $K_i \models g_i(head(r_i))$, then there is no extension g'_i of h'_i such that $J_i \models g'_i(head(r'_i))$.

To show that $\Sigma' \in \mathsf{CT}^\star_\exists$ implies $\Sigma \in \mathsf{CT}^\star_\exists$ we can follow an approach analogous to the one described above. Thus, given a database D and terminating $\star$-chase sequence S' of D with Σ', we can build from S' a terminating $\star$-chase sequence of D with Σ. The construction is analogous to the one discussed above, but it proceeds the other way around.

2. Consider the set of dependencies $\Sigma_{4.48}$ of Example 4.48 and the two sets of TGDs which can be obtained by means of the substitution-free simulation (see Example 4.48), say $\Sigma'_{4.48}$ and $\Sigma''_{4.48}$. It can be easily verified that for every database D, all $\star$-chase sequences of D with $\Sigma_{4.48}$ are terminating. Thus, $\Sigma_{4.48}$ belongs to both $\mathsf{CT}^\star_\forall$ and $\mathsf{CT}^\star_\exists$. On the other hand, there is no terminating $\star$-chase sequence of the database $D = \{c(a)\}$ with $\Sigma'_{4.48}$ or $\Sigma''_{4.48}$. Hence, $\Sigma'_{4.48}$ (resp. $\Sigma''_{4.48}$) belongs to neither $\mathsf{CT}^\star_\exists$ nor $\mathsf{CT}^\star_\forall$. □

Proof of Theorem 4.55

Consider the directed graph G obtained from the activation graph $G_f(\Sigma)$ by replacing each strongly connected component by a single vertex. Clearly, G is acyclic. By definition of semi-stratification, each vertex v in G is associated with a weakly acyclic set of dependencies, which we denote as Σ_v.

Notice that since each Σ_v is weakly acyclic, then for every instance K, every standard Chase sequence of K with Σ_v has a length polynomial in the size of K [38]; for every instance K, every standard Chase sequence of K with a set of full dependencies has a length that is polynomial in the size of K [38].

From the two aforementioned properties it follows that, for every Σ_v, for every instance K, every standard Chase sequence of K with $\Sigma_v \cup \Sigma_\forall$ has a length that is polynomial in the size of K.

Consider now the standard Chase sequence S defined as follows. Let $v_1, ..., v_n$ be a topological ordering of G. We define $K_0 = D$ and for $1 \leq i \leq n$, K_i is the result of a terminating standard Chase sequence of K_{i-1} with $\Sigma_v \cup \Sigma_\forall$ where $\Sigma_\forall$ is exhaustively enforced before a Chase step with a dependency in Σ_{v_i} is performed.

From the observations made before, it follows that the length of S is polynomial in the size of D. Furthermore, S is terminating, because otherwise there would exists an edge from a dependency of v_i to a dependency of v_j with $j < i$. □

Proof of Theorem 4.56

First of all, given two dependencies r_1 and r_2, deciding whether $r_1 < r_2$ is in NP. In fact, we can guess K, J, h_1, and h_2 as per Definition 4.52 and verify if the conditions of Definition 4.52 are satisfied. Similar to the proof of Theorem 3 of [33], K and J are bounded by $|r_1|$ and $|r_2|$. Thus, to check if a set of dependencies Σ is not stratified, we can guess a strongly connected component of $G_f(\Sigma)$ and verify that is not weakly acyclic. □

Proof of Theorem 4.57

1. We start by showing $\mathcal{S}tr \subseteq \mathcal{S}\text{-}\mathcal{S}tr$. Consider a set Σ of dependencies in $\mathcal{S}tr$, and let $G(\Sigma) = (\Sigma, E)$ and $G_f(\Sigma) = (\Sigma, E_f)$. By definition of G and G_f, it is easy to see that $E_f \subseteq E$. Thus, every strongly connected component of

$G_f(\Sigma)$ is a cycle of G, and since Σ is stratified, then the strongly connected component is weakly acyclic. Hence, Σ is semi-stratified.
To prove proper containment, it suffices to notice that $\Sigma_{4.54}$ of Example 4.54 belongs to $\mathcal{S}$-$\mathcal{S}tr$ but is not in $\mathcal{S}tr$.

2. To prove that $\mathcal{C} \nsubseteq \mathcal{S}$-$\mathcal{S}tr$, for $\mathcal{C} \in \{\mathcal{SC}, \mathcal{AC}, \mathcal{MFA}\}$, consider the following set of dependencies Σ:

$$N(x) \rightarrow \exists y\ E(x,y)$$
$$S(y) \wedge E(x,y) \rightarrow N(y)$$

It can be easily verified that $\Sigma \in \mathcal{SC}$, and thus Σ belongs also to $\mathcal{AC}$ and $\mathcal{MFA}$, but $\Sigma \notin \mathcal{S}$-$\mathcal{S}tr$.
To prove that $\mathcal{S}$-$\mathcal{S}tr \nsubseteq \mathcal{C}$, for $\mathcal{C} \in \{\mathcal{SC}, \mathcal{AC}, \mathcal{MFA}\}$, it suffices to notice that $\Sigma_{4.54}$ of Example 4.54 belongs to $\mathcal{S}$-$\mathcal{S}tr$ but is not in $\mathcal{C}$, as $\Sigma_{4.54} \notin \mathsf{CT}_\forall^{\mathsf{std}}$ and $\mathcal{C} \subsetneq \mathsf{CT}_\forall^{\mathsf{std}}$. □

Proof of Theorem 4.60

First of all, notice that once a rule body is adorned in a certain way, the same adorned body is not used again for that rule. Suppose by contradiction that Algorithm $Adn^{\exists}$ does not terminate for a set of dependencies Σ. Thus there is an infinite number of adorned dependencies that are added to Σ^{μ}. Since adorned dependencies are derived by adorning dependencies in Σ whose cardinality is finite, then there must be an infinite number of adorned dependencies derived from a dependency $r \in \Sigma$. Then, there must be at least one position where an infinite number of adornment symbols is introduced. Thus, there must be two adorned dependencies s.t. one can be mapped to the other, which is a contradiction. □

Proof of Theorem Equivalence-Theorem-v2

1. We show that for every standard Chase sequence $S^{\mu} = K_0^{\mu} \xrightarrow{r_0^{\mu}, h_0, \gamma_0} K_1^{\mu} \xrightarrow{r_1^{\mu}, h_1, \gamma_1} \cdots$ of K_0^{μ} with Σ^{μ}, where $K_0^{\mu} = D$, there is a standard Chase sequence $S = K_0 \xrightarrow{r_0, h_0, \gamma_0} K_1 \xrightarrow{r_1, h_1, \gamma_1} \cdots$ of D with Σ, where $K_0 = D$, such that for every $i \geq 0$ it is the case that $K_i = src(K_i^{\mu})$ (modulo renaming of labeled nulls) and $r_i = src(r_i^{\mu})$.

It can be easily verified by induction on i ($i \geq 0$) that if $K_i^\mu \xrightarrow{r_i^\mu, h_i, \gamma_i} K_{i+1}^\mu$ is a Chase step such that there is no extension h_i' of h_i such that $K_i^\mu \models h_i'(head(r_i^\mu))$, then $K_i \xrightarrow{r_i, h_i, \gamma_i} K_{i+1}$ is a Chase step where there is no extension h_i' of h_i such that $K_i \models h_i'(head(r_i))$ and, $K_i = src(K_i^\mu)$, $K_{i+1} = src(K_{i+1}^\mu)$ and $r_i = src(r_i^\mu)$.

2. From the previous result we have that $CMod(D, \Sigma^\mu) \neq \varnothing$ implies $CMod(D, \Sigma) \neq \varnothing$. Moreover, if $CMod(D, \Sigma) \neq \varnothing$ there must be a terminating standard Chase sequence $S = K_0 \xrightarrow{r_0, h_0, \gamma_0} K_1 \xrightarrow{r_1, h_1, \gamma_1} \cdots$ of $D = K_0$ with Σ, where full dependencies are applied, whenever possible, before existentially quantified ones. Moreover, this also implies that there must be a standard Chase sequence $S^\mu = K_0^\mu \xrightarrow{r_0^\mu, h_0, \gamma_0} K_1^\mu \xrightarrow{r_1^\mu, h_1, \gamma_1} \cdots$ of $K_0 = D$ with Σ^μ (where full dependencies are applied, whenever possible, before existentially quantified ones) such that $src(K_i^\mu) = K_i$. Indeed, for any Chase step $K_i \xrightarrow{r_i, h_i, \gamma_i} K_{i+1}$, if there is an instance K_i^μ such that $src(K_i^\mu) = K_i$, then the adorned atoms in K_i^μ have been derived by applying dependencies in Σ^μ (namely dependencies $r_0^\mu, ..., r_{i-1}^\mu$ with homomorphisms $h_0, ..., h_{i-1}$, respectively). But this means that these dependencies, occurring in Σ^μ, have been used to adorn r_i obtaining the adorned dependency r_i^μ; this dependency can be applied with the same homomorphism h_i and, consequently, we derive the instance K_{i+1}^μ with $src(K_{i+1}^\mu) = K_{i+1}$. □

Proof of Theorem 4.63

First of all, observe that if the variable *Acyc* is *true*, then the adorned program is not recursive, that is, there is no cyclic sequence of rules which can be executed indefinitely. This is ensured by the test performed on line 15 of Algorithm 4.6, which checks if an adornment symbol is cyclic. Therefore, the number of atoms which can be derived is polynomial in the size of the input database, as constants can appear only in arguments adorned with b, and it is linear in the number of free symbols introduced in the adorned program, which does not depend on the considered database. □

Proof of Theorem 4.64

($\mathcal{S}$-$\mathcal{S}tr \subsetneq \mathcal{SAC}$). We show that $\Sigma \notin \mathcal{QAC}$ implies $\Sigma \notin \mathcal{S}$-$\mathcal{S}tr$, for every set of dependencies Σ. In fact, $\Sigma \notin \mathcal{QAC}$ iff the condition on line 15 of Algorithm 4.6

is verified. This means that there is an existentially quantified dependency r whose head is cyclic because of the adornment symbol associated with a position $p[i]$, that is with the i-th term of a p-atom in the head. By definition of cyclic adornment symbol, the activation graph $G_f(\Sigma)$ has a strongly connected component C containing r which is not weakly acyclic because of a cycle involving $p[i]$ with a special edge (incoming in $p[i]$). Consequently, $\Sigma \notin \mathcal{S}\text{-}\mathcal{S}tr$.

It can be easily verified that the following set of dependencies is in $\mathcal{SAC}$ but not in $\mathcal{S}\text{-}\mathcal{S}tr$, whence proper containment.

$$n(X) \rightarrow \exists Y\ e(X,Y)$$
$$s(X) \wedge E(x,y) \rightarrow n(Y)$$

($\mathcal{AC} \subsetneq \mathcal{SAC}$). We show that if Algorithm 4.6 sets the *Acyc* variable to *false*, so does the algorithm employed by $\mathcal{AC}$. To this end, notice that the order followed by Algorithm 4.6 when adorning dependencies is a valid order for the algorithm employed by $\mathcal{AC}$. This is still the case if substitution-free simulation is applied before calling the algorithm employed by $\mathcal{AC}$. However, the two algorithms use different activation relations. Nevertheless, notice that, given two dependencies r_1 and r_2, if $r_1 < r_2$ holds (cf. Definition 4.52), then r_1 fires r_2 according to the activation relation of $\mathcal{AC}$ (cf. Definition 4 of [56]). Thus, if Algorithm 4.6 set the *Acyc* variable to *false*, the algorithm employed by $\mathcal{AC}$ must do the same.

The set of dependencies $\Sigma_{1.2}$ of Example 1.2 shows proper containment: as shown in Example 4.58, $\Sigma_{1.2} \in \mathcal{SAC}$, but $\Sigma_{1.2}$ cannot belong to $\mathcal{AC}$ as $\Sigma_{1.2} \notin \mathsf{CT}_{\forall}^{\mathsf{std}}$. □

Proof of Theorem 4.65

Let $\Sigma^{\mu} = Adn^{\exists}(\Sigma)[1]$. Suppose $\Sigma \in Adn^{\exists}\text{-}\mathcal{C}$, that is, $\Sigma^{\mu} \in \mathcal{C}$. Then, Σ^{μ} belongs to $\mathsf{CT}_{\exists}^{\mathsf{std}}$ or $\mathsf{CT}_{\forall}^{\mathsf{std}}$ (depending on criterion C). In both cases, for every database D, $CMod(D, \Sigma^{\mu}) \neq \varnothing$. By Theorem 4.61, $CMod(D, \Sigma) \neq \varnothing$. Hence, $\Sigma \in \mathsf{CT}_{\exists}^{\mathsf{std}}$. □

Proof of Theorem 4.66

All the criteria considered in Section 4.2 analyze relations between dependencies and propagation of arguments among positions. These relations are based

on the structural properties of a set of dependencies. Assume by contradiction that there exist a set of dependencies Σ and a criterion $C \in \mathcal{C}$ such that $\Sigma \in \mathcal{C}$ and $\Sigma^{\mu} = Adn^{\exists}\exists(\Sigma)[1]$ does not belong to $\mathcal{C}$. This means that either *(1)* there are two dependencies $r, s \in \Sigma$, two adorned dependencies $r^{\alpha}, s^{\beta} \in \Sigma^{\mu}$ and a relation θ, such that r^{α} and s^{β} satisfy θ, whereas r and s does not, or *(2)* there exists a labelled null that can be propagated from a position i of $head(r^{\alpha})$ to a position j of $head(s^{\beta})$, whereas a labelled null cannot be propagated from position i of $head(r)$ to position j of $head(s)$. Clearly, this is not possible by construction of Σ^{μ}.

To show proper containment, we must show that for each $C \in \mathcal{C}$, there is a set of dependencies that belongs to $Adn^{\exists}$-$\mathcal{C}$ and do not belong to $\mathcal{C}$. Consider the set of dependencies $\Sigma_{4.66}$ consisting of the following two dependencies:

$$n(X) \rightarrow \exists Y\ e(X,Y)$$
$$s(X) \wedge E(x,y) \rightarrow n(Y)$$

It has been shown that $\Sigma \notin \mathcal{LS}$ [56] and $\Sigma \notin \mathcal{MFA}$ [50]. As a consequence, Σ does not belong to any of the classes included in $\mathcal{LS}$ or $\mathcal{MFA}$. However, $\Sigma^{\mu} = Adn^{\exists}(\Sigma)[1]$ is not recursive and thus recognized by all methods considered in Section 4.2. The only criterion that remains to be considered is AC. In this regard, consider the set of dependencies $\Sigma_{4.48}$ of Example 4.48: it is not recognized as terminating by AC (even considering the set of TGDs obtained by the substitution-free simulation method); however, $\Sigma_{4.48} \in Adn^{\exists}$-$\mathcal{AC}$. □

Proof of Theorem 4.67

Given a set of dependencies Σ and a dependency $r \in \Sigma$, we denote by

- $uv(r)$ the number of distinct universally quantified variables in r,
- $ev(r)$ the number of distinct existentially quantified variables in r,
- $v(r) = uv(r) + ev(r)$ the number of distinct variables in r,
- $muv(\Sigma) = max\{\ uv(r) \mid r \in \Sigma\}$,
- $mev(\Sigma) = max\{\ ev(r) \mid r \in \Sigma\}$.
- $mv(\Sigma) = max\{\ v(r) \mid r \in \Sigma\}$.

We start by computing the cardinality of the set of adorned dependencies. Such a set satisfies the condition that for each two adorned dependencies derived from the same dependency $r \in \Sigma$, say them r^{α} and r^{β} with $\alpha = \alpha_1 \ldots \alpha_n$ and $\beta = \beta_1 \ldots \beta_n$, there is no valid unifier. This can happens because

there are two adornment definitions $\alpha_i = f_y^u(\ldots)$ and $\beta_i = f_Z^v(\ldots)$ with $f_Y^u \neq f_Z^v$, i.e. $u \neq v$ or $u = v$ and $Y \neq Z$. Let α be an adornment for r, we can have $O(ev(\Sigma)^{v(r)})$ different adornments which could not unify with α, as the adornments are associated with at most $ev(\Sigma)$ different function symbols.

Therefore the number of different adornments for r is bounded by $O\left(ev(\Sigma)^{v(r)}\right)$, the global number of existential adorned dependencies is bounded by

$$O\left(\sum_{r\in\Sigma}\left(ev(\Sigma)^{v(r)}\right)\right) = O\left(|\Sigma| \times ev(\Sigma)^{mv(\Sigma)}\right)$$

Let $F(\Sigma)$ be the set of adornment symbols which can be generated by Algorithm 4.6, the number of free adornment symbols $|F(\Sigma)|$ is bounded by

$$|F(\Sigma)| = O\left(|\Sigma| \times ev(\Sigma)^{mv(\Sigma)} \times mev(\Sigma)\right)$$

$$= O\left(|\Sigma| \times ev(\Sigma)^{mv(\Sigma)}\right)$$

as in each dependency we can introduce at most $mev(\Sigma_E)$ new adornments.

The size of Σ^γ is bounded by

$$O\left(|F(\Sigma)|^{mv(\Sigma)}\right) = O\left(\left(|\Sigma| \times ev(\Sigma)^{mv(\Sigma)}\right)^{mv(\Sigma)}\right) =$$

$$O\left(|\Sigma|^{mv(\Sigma)} \times ev(\Sigma)^{mv(\Sigma)^2}\right)$$

and, therefore, exponential in the maximal number of variables occurring in a rule and in the number of rules.

Considering the time complexity, analogous to the stratification checking problem, the problem of checking whether a dependency is fireable (function *adorn*) is in *NP*. As the *NP* problem can be solved with a deterministic Turing machine with exponential time complexity in the size of the adorned set of dependencies Σ^μ, the time complexity is double exponential in the size of the input set of dependencies Σ. □

References

1. Information Systems Group Ontologies, `http://www.cs.ox.ac.uk/isg/ontologies/`.
2. The OBO Foundry, `http://www.obofoundry.org`.
3. Phenoscape Ontologies, `http://phenoscape.org/wiki/Ontologies`.
4. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
5. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
6. M. Alviano, W. Faber, and N. Leone. Disjunctive ASP with functions: Decidable queries and effective computation. *TPLP*, 10(4-6):497–512, 2010.
7. Marcelo Arenas, Pablo Barceló, Ronald Fagin, and Leonid Libkin. Locally consistent transformations and query answering in data exchange. In *PODS*, pages 229–240, 2004.
8. Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *PODS*, pages 68–79, 1999.
9. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *TCS*, 236(1-2):133–178, 2000.
10. Franz Baader. Least common subsumers and most specific concepts in a description logic with existential restrictions and terminological cycles. In *IJCAI*, pages 319–324, 2003.
11. Jean-Francois Baget, Michel Leclère, Marie-Laure Mugnier, and Eric Salvat. On rules with existential variables: Walking the decidability line. *Artif. Intell.*, 175(9-10):1620–1654, 2011.
12. S. Baselice, P. A. Bonatti, and G. Criscuolo. On finitely recursive programs. *TPLP*, 9(2):213–238, 2009.
13. Catriel Beeri and Moshe Y. Vardi. Formal systems for tuple and equality generating dependencies. *SIAM J. Comput.*, 13(1):76–98, 1984.

14. Leopoldo E. Bertossi, Solmaz Kolahi, and Laks V. S. Lakshmanan. Data cleaning and query answering with matching dependencies and matching functions. In *ICDT*, 2011.
15. M. Bruynooghe, M. Codish, J. P. Gallagher, S. Genaim, and W. Vanhoof. Termination analysis of logic programs through combination of type-based norms. *ACM TOPLAS*, 29(2), 2007.
16. M. Calautti, S. Greco, C. Molinaro, and I. Trubitsyna. Checking termination of logic programs with function symbols through linear constraints. In *RuleML*, pages 97–111, 2014.
17. M. Calautti, S. Greco, F. Spezzano, and I. Trubitsyna. Checking termination of bottom-up evaluation of logic programs with function symbols. *TPLP*, 2014.
18. M. Calautti, S. Greco, and I. Trubitsyna. Detecting decidable classes of finitely ground logic programs with function symbols. In *PPDP*, 2013.
19. Marco Calautti, Georg Gottlob, and Andreas Pieris. Chase termination for guarded existential rules. In *PODS*, 2015.
20. Marco Calautti, Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. Exploiting equality generating dependencies in checking chase termination. *VLDB*, 2015.
21. Marco Calautti, Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. Logic program termination analysis using atom sizes. In *IJCAI*, pages 2833–2839, 2015.
22. Andrea Cali, Georg Gottlob, and Michael Kifer. Taming the infinite chase: Query answering under expressive relational constraints. *J. Artif. Intell. Res.*, 48:115–174, 2013.
23. Andrea Cali, Georg Gottlob, and Thomas Lukasiewicz. A general Datalog-based framework for tractable query answering over ontologies. *J. Web Sem.*, 14:57–83, 2012.
24. Andrea Cali, Georg Gottlob, and Andreas Pieris. Advanced processing for ontological queries. *PVLDB*, 3(1):554–565, 2010.
25. Andrea CalÌ, Georg Gottlob, and Andreas Pieris. Towards more expressive ontology languages: The query answering problem. *Artif. Intell.*, 193:87–128, 2012.
26. F. Calimeri, S. Cozza, G. Ianni, and N. Leone. Computable functions in ASP: Theory and implementation. In *ICLP*, pages 407–424, 2008.
27. Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable reasoning and efficient query answering in description logics: The DL-Lite family. *J. Autom. Reasoning*, 39(3):385–429, 2007.
28. J. Chomicki. *A decidable class of logic programs with function symbols.* Manhattan, Kan: Kansas State University, Dept. of Computing and Information Sciences, 1990.
29. J. Chomicki and T. Imieliński. Finite representation of infinite query answers. *ACM Trans. Database Syst.*, 18(2):181–223, June 1993.

30. M. Codish, V. Lagoon, and P. J. Stuckey. Testing for termination with monotonicity constraints. In *ICLP*, pages 326–340, 2005.
31. Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. On reconciling data exchange, data integra- tion, and peer data management. In *PODS*, 2007.
32. D. De Schreye and S. Decorte. Termination of logic programs: The never-ending story. *JLP*, 19/20:199–260, 1994.
33. Alin Deutsch, Alan Nash, and Jeffrey B. Remmel. The chase revisited. In *PODS*, pages 149–158, 2008.
34. T. Eiter, M. Fink, T. Krennwallner, and C. Redl. Liberal safety for answer set programs with external sources. In *AAAI*, 2013.
35. T. Eiter and M. Simkus. Fdnc: Decidable nonmonotonic disjunctive logic programs with function symbols. *ACM TOCL*, 11(2), 2010.
36. J. Endrullis, J. Waldmann, and H. Zantema. Matrix interpretations for proving termination of term rewriting. *JAR*, 40(2-3):195–220, 2008.
37. Ronald Fagin. Equality-generating dependencies. In *Encyclopedia of Database Systems*. 2009.
38. Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *Th. Comp. Sc.*, 336(1):89–124, 2005.
39. Ronald Fagin, Phokion G. Kolaitis, and Lucian Popa. Data exchange: getting to the core. *ACM TODS*, 30(1):174–210, 2005.
40. Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM TODS*, 33(2), 2008.
41. S. Flesca, F. Furfaro, and F. Parisi. Querying and repairing inconsistent numerical databases. *ACM TODS*, 35(2), 2010.
42. Tom Gardiner, Dmitry Tsarkov, and Ian Horrocks. Framework for an automated comparison of description logic reasoners. In *ISWC*, pages 654–667, 2006.
43. M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on AI and ML. Morgan & Claypool Publishers, 2012.
44. M. Gebser, T. Schaub, and S. Thiele. Gringo: A new grounder for answer set programming. In *LPNMR*, pages 266–271, 2007.
45. Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. The LLUNATIC data-cleaning framework. *PVLDB*, 2013.
46. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080, 1988.
47. T. Gogacz and J. Marcinkowski. All-instances termination of chase is undecidable. In *ICALP*, 2014.
48. Georg Gottlob and Alan Nash. Efficient core computation in data exchange. *J. ACM*, 55(2), 2008.

49. G"osta Grahne and Adrian Onet. Anatomy of the chase. *CoRR*, abs/1303.6682, 2013.
50. Bernardo Cuenca Grau, Ian Horrocks, Markus Krotzsch, Clemens Kupke, Despoina Magka, Boris Motik, and Zhe Wang. Acyclicity notions for existential rules and their application to query answering in ontologies. *JAIR*, 47:741–808, 2013.
51. S. Greco, C. Molinaro, and F. Spezzano. *Incomplete Data and Data Dependencies in Relational Databases.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2012.
52. S. Greco, C. Molinaro, and I. Trubitsyna. Bounded programs: A new decidable class of logic programs with function symbols. In *IJCAI*, 2013.
53. S. Greco, C. Molinaro, and I. Trubitsyna. Logic programming with function symbols: Checking termination of bottom-up evaluation through program adornments. *TPLP*, 13(4-5):737–752, 2013.
54. S. Greco and F. Spezzano. Chase termination: A constraints rewriting approach. *Proceeding of the Very Large Data Base Conference*, 3(1):93–104, 2010.
55. Sergio Greco, Francesca Spezzano, and Irina Trubitsyna. Stratification criteria and rewriting techniques for checking chase termination. *PVLDB*, 4(11):1158–1168, 2011.
56. Sergio Greco, Francesca Spezzano, and Irina Trubitsyna. Checking chase termination: Cyclicity analysis and rewriting techniques. *IEEE TKDE*, 27(3):621–635, 2015.
57. Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.
58. André Hernich and Nicole Schweikardt. CWA-solutions for data exchange settings with target dependencies. In *PODS*, pages 113–122, 2007.
59. Philip M. Lewis, Richard Edwin Stearns, and Juris Hartmanis. Memory bounds for recognition of context-free and context-sensitive languages. In *FOCS*, pages 191–202, 1965.
60. Y. Lierler and V. Lifschitz. One more decidable class of finitely ground programs. In *ICLP*, pages 489–493, 2009.
61. David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. Testing implications of data dependencies. *ACM TODS*, 4(4):455–469, 1979.
62. M. Marchiori. Proving existential termination of normal logic programs. In *AMST*, 1996.
63. B. Marnette. Generalized schema-mappings: from termination to tractability. In *PODS*, pages 13–22, 2009.
64. M. Meier, M. Schmidt, and G. Lausen. On chase termination beyond stratification. *CoRR*, abs/0906.4228, 2009.
65. Michael Meier. *On the Termination of the Chase Algorithm.* Albert-Ludwigs-Universitat Freiburg, 2010.

66. Michael Meier, Michael Schmidt, and Georg Lausen. On chase termination beyond stratification. *PVLDB*, 2(1):970–981, 2009.
67. M. Thang Nguyen, J. Giesl, P. Schneider-Kamp, and D. De Schreye. Termination analysis of logic programs based on dependency graphs. In *LOPSTR*, pages 8–22, 2007.
68. E. Ohlebusch. Termination of logic programs: Transformational methods revisited. *AAECC*, 12(1/2):73–116, 2001.
69. Adrian Onet. The chase procedure and its applications in data exchange. In *Data Exchange, Integration, and Streams*, pages 1–37. 2013.
70. C. H. Papadimitriou. On the complexity of integer programming. *Journal of the ACM*, 28(4):765–768, 1981.
71. F. Riguzzi and T. Swift. Well-definedness and efficient inference for probabilistic logic programming under the distribution semantics. *TPLP*, 13(2):279–302, 2013.
72. F. Riguzzi and T. Swift. Terminating evaluation of logic programs with finite three-valued models. *ACM TOCL*, 2014.
73. P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated termination proofs for logic programs by term rewriting. *ACM TOCL*, 11(1), 2009.
74. Richard Edwin Stearns, Juris Hartmanis, and Philip M. Lewis. Hierarchies of memory limited computations. In *FOCS*, pages 179–190, 1965.
75. C. Sternagel and A. Middeldorp. Root-labeling. In *RTA*, pages 336–350, 2008.
76. T. Syrjanen. Omega-restricted logic programs. In *LPNMR*, pages 267–279, 2001.
77. J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.
78. M. Venturini Zilli. Complexity of the unification algorithm for first-order expressions. *CALCOLO*, 12(4):361–371, 1975.
79. S. Verbaeten, D. De Schreye, and K. F. Sagonas. Termination proofs for logic programs with tabling. *ACM TOCL*, 2(1):57–92, 2001.
80. D. Voets and D. De Schreye. Non-termination analysis of logic programs with integer arithmetics. *TPLP*, 11(4-5):521–536, 2011.