University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

SON THANH CAO

METHODS FOR EVALUATING QUERIES TO HORN KNOWLEDGE BASES IN FIRST-ORDER LOGIC

PhD dissertation

Supervisors:

dr hab. Linh Anh Nguyen
Institute of Informatics
University of Warsaw

dr. Joanna Golińska-Pilarek
Institute of Philosophy
University of Warsaw

August, 2015

Author's declaration:

Aware of legal responsibility I hereby declare that I have written this dissertation myself and all its contents have been obtained by legal means.

. . . .	. . . .
date	*Son Thanh Cao*

Supervisors' declaration:

The dissertation is ready to be reviewed.

. . . .	. . . .
date	*dr hab. Linh Anh Nguyen*
. . . .	. . . .
date	*dr. Joanna Golińska-Pilarek*

Abstract

Horn knowledge bases are extensions of Datalog deductive databases without the range-restrictedness and function-free conditions. A Horn knowledge base consists of a positive logic program for defining intensional predicates and an instance of extensional predicates. This dissertation concentrates on developing efficient methods for evaluating queries to Horn knowledge bases. In addition, a method for evaluating queries to stratified knowledge bases is also investigated. This topic has not been well studied as query processing for Datalog-like deductive databases or the theory and techniques of logic programming.

We begin with formulating query-subquery nets and use them to create the first framework for developing algorithms for evaluating queries to Horn knowledge bases with the following good properties: the approach is goal-directed; each subquery is processed only once; each supplement tuple, if desired, is transferred only once; operations are done set-at-a-time; and any control strategy can be used. Our intention is to increase efficiency of query processing by eliminating redundant computation, increasing adjustability (i.e., easiness in adopting advanced control strategies) and reducing the number of accesses to the secondary storage. The framework forms a generic evaluation method called QSQN. It is sound and complete, and has polynomial time data complexity when the term-depth bound is fixed.

Next, we incorporate tail-recursion elimination into query-subquery nets in order to formulate the QSQN-TRE evaluation method for Horn knowledge bases. The aim is to reduce materializing the intermediate results during the processing of a query with tail-recursion. We prove the soundness and completeness of the proposed method and show that, when the term-depth bound is fixed, the method has polynomial time data complexity. We then extend QSQN-TRE to obtain another evaluation method called QSQN-rTRE, which can eliminate not only tail-recursive predicates but also intensional predicates that appear rightmost in the bodies of the program clauses.

We also incorporate stratified negation into query-subquery nets to obtain a method called QSQN-STR for evaluating queries to stratified knowledge bases.

We propose the control strategies DAR, DFS, IDFS and implement the methods QSQN, QSQN-TRE, QSQN-rTRE together with these strategies. Then, we carry out experiments to obtain a comparison between these methods (using the IDFS control strategy) and the other well-known evaluation methods such as Magic-Sets and QSQR. We also report experimental results of QSQN-STR using a control strategy called IDFS2, which is a modified version of IDFS. The experimental results confirm the efficiency and usefulness of the proposed evaluation methods.

Keywords: Horn knowledge bases, stratified knowledge bases, deductive databases, logic programming, query processing, query optimization, magic-sets transformation, query-subquery recursive, tail-recursion elimination, Datalog.

ACM Computing Classification System: H.2.4 (*Query Processing, Query Optimization, Rule-based Databases*), D.1.6 (*Logic Programming*).

Streszczenie[1]

Bazy wiedzy typu Horna są uogólnieniem dedukcyjnych baz danych Datalogu bez ograniczeń o zakresie zmiennych i z możliwością korzystania z symboli funkcyjnych. Baza wiedzy typu Horn składa się z pozytywnego programu w logice definiującego predykaty intensjonalne i instancji ekstensjonalnych predykatów. Niniejsza rozprawa dotyczy efektywnych metod obliczania zapytań do baz wiedzy typu Horna. Omówiona jest również metoda obliczania zapytań do stratyfikowanych baz wiedzy. Problematyka ta nie była do tej pory tak dobrze zbadana, jak przetwarzanie zapytań dla dedukcyjnych baz danych czy teoria i techniki programowania w logice.

W pierwszej części rozprawy formułujemy sieci zapytań-podzapytań i omawiamy konstrukcję bazującej na takich sieciach metody obliczania zapytań do baz wiedzy typu Horna, o następujących dobrych własnościach: zastosowane podejście jest zorientowane na cel; każde podzapytanie jest przetwarzane tylko raz; każda krotka uzupełniająca jest przesyłana tylko raz, o ile jest to pożądane; operacje są wykonywane zbiorowo; każda strategia sterowania może być używana. Intencją tej metody jest zwiększenie efektywności przetwarzania zapytań poprzez wyeliminowanie zbędnych obliczeń, ułatwienie stosowania zaawansowanych strategii sterowania oraz zredukowanie liczby odczytów i zapisów dyskowych. Ogólna taka metoda jest nazwana QSQN. Jest ona poprawna i pełna oraz ma złożoność wielomianową względem danych ekstensjonalnych, o ile głębokość zagnieżdżenia termów jest ograniczona.

W dalszej części rozprawy przedstawiona jest technika włączania eliminacji rekurencji ogonowej do sieci zapytań-podzapytań i uzyskana w ten sposób metoda obliczania zapytań QSQN-TRE dla baz wiedzy typu Horna. Celem takiej eliminacji jest redukcja zachowywania wyników pośrednich podczas przetwarzania zapytań z rekurencją ogonową. Udowodniono, że metoda QSQN-TRE jest poprawna i pełna oraz ma złożoność wielomianową względem danych ekstensjonalnych, o ile głębokość zagnieżdżenia termów jest ograniczona. Jako rozszerzenie metody QSQN-TRE została opracowana również inna metoda obliczania zapytań o nazwie QSQN-rTRE, która pozwala wyeliminować nie tylko predykaty ogonowo rekurencyjne, ale również predykaty intensjonalne, występujące na końcu ciała pewnej klauzuli programu.

Opracowane zostały również sieci zapytań-podzapytań i odpowiednia metoda o nazwie QSQN-STR do obliczania zapytań do stratyfikowanych baz wiedzy. Takie bazy wiedzy umożliwiają użycie bezpiecznych literałów negatywnych w ciałach klauzul programu.

Metody QSQN, QSQN-TRE i QSQN-rTRE zostały zaimplementowane z trzema zaproponowanymi strategiami sterowania DAR, DFS i IDFS. Przeprowadzone zostały eksperymenty mające na celu porównanie tych metod (używających strategii sterowania IDFS) z innymi znanymi metodami obliczania zapytań, takimi jak Magic-Sets i QSQR. Omówione zostały również wyniki eksperymentów działania metody QSQN-STR ze strategią sterowania IDFS2 będącą zmodyfikowaną wersją IDFS. Wyniki przeprowadzonych eksperymentów potwierdzają skuteczność i przydatność opracowanych metod obliczania zapytań.

[1]The abstract and keywords have been translated from English to Polish by the supervisors.

Słowa kluczowe: Bazy wiedzy typu Horna, stratyfikowane bazy wiedzy, dedukcyjne bazy danych, programowanie w logice, przetwarzanie zapytań, optymalizacja obliczania zapytań, transformacja magic-sets, QSQR, eliminacja rekurencji ogonowej, Datalog.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisors, dr hab. Linh Anh Nguyen and dr. Joanna Golińska-Pilarek, from the University of Warsaw for their encouragement, patience and support over the years. Both of them were always ready to give me instructions, discuss scientific problems, share their experience and exchange new ideas throughout the course of my research. This dissertation would not be possible without their help and guidance. I have learnt many things from them and I am inspired by their love for the research work.

I am sincerely grateful to Professor Andrzej Szałas for sharing his wisdom and illuminating views on a number of issues related to my research.

I am very much thankful to the Faculty of Mathematics, Informatics and Mechanics, University of Warsaw (MIMUW) and the Warsaw Center of Mathematics and Computer Science (WCMCS) for accepting me to the PhD study at MIMUW and giving me a fellowship of WCMCS. The fellowship was essential for my stay in Poland.

I would like to thank the secretaries of the Faculty of MIMUW, especially Marlena Nowińska and Maria Gamrat for their help in many different ways and handling the paperwork on cases.

I would like to thank my colleagues at the Faculty of Information Technology, Vinh University, who have granted me the necessary time for my PhD study. Especially, many thanks to dr. Phan Anh Phong for very useful comments and suggestions throughout my work and studies, and to Tran Thi Kim Oanh for allowing me to use her laptop and for very helpful assistance.

I am very grateful to my friends, old and new, for keeping in touch, being interested in my work and sharing experiences during my stay in Poland.

Last but not least, I would like to express my special thanks to my parents, my wife, my daughter and the other family members for their love, encouragement and advice. They were always supportive and encouraged me with their best wishes. I love them all.

This work was supported by Polish National Science Centre (NCN) under Grant No. 2011/02/A/HS1/00395.

Contents

Chapter 1

Introduction

Query processing is an important research area in computer science and information technology. Interest in deductive databases and methods for evaluating Datalog or Datalog$^\neg$ queries intensified in the eighties and early nineties, but "a perceived lack of compelling applications at the time ultimately forced Datalog research into a long dormancy" [33]. As also observed by Huang et al. in their SIGMOD'2011 paper [33]:

> *"We are witnessing an exciting revival of interest in recursive Datalog queries in a variety of emerging application domains such as data integration, information extraction, networking, program analysis, security, and cloud computing. [...]*
>
> *As the list of applications above indicates, interest today in Datalog extends well beyond the core database community. Indeed, the successful Datalog 2.0 Workshop held in March 2010 at Oxford University attracted over 100 attendees from a wide range of areas (including databases, programming languages, verification, security, and AI)."*

During the last decade, rule-based query languages, including languages related to Datalog, were also intensively studied for the Semantic Web (e.g., in [5, 10, 20, 21, 26, 27, 36, 39, 40, 52, 54]). In general, since deductive databases and knowledge bases are widely used in practical applications, improvements for processing recursive queries are always desirable. Due to the importance of the topic, it is worth doing further research on the topic.

Horn knowledge bases are extensions of Datalog deductive databases without the range-restrictedness and function-free conditions [1]. As argued in [39], the Horn fragment of first-order logic plays an important role in knowledge representation and reasoning. A Horn knowledge base consists of a positive logic program for defining intensional predicates and an instance of extensional predicates. When the knowledge base is too big, not all of the extensional and intensional relations may be totally kept in the computer memory and query evaluation may not be totally done in the computer memory. In such cases, the system usually has to load (resp. unload) relations from (resp. to) the secondary storage. Thus, in contrast to logic programming, for Horn knowledge bases efficient access to the secondary storage is a very important aspect.

This dissertation studies query processing for Horn knowledge bases. Particularly, we concentrate on developing efficient methods for evaluating queries to Horn knowledge bases. In addition, query evaluation for stratified knowledge bases is also investigated. This topic has not been well studied as query processing for the Datalog-like deductive databases or the theory and techniques of logic programming.

1.1 Related Work

This section discusses related work on evaluation methods for Datalog databases and Horn knowledge bases. The survey [50] by Ramakrishnan and Ullman provides a good overview of deductive database systems by 1995, with a focus on implementation techniques. The book [1] by Abiteboul et al. is also a good source for references. We present here only a brief overview of the subject, which is based on [1, 39, 45].

In [69], Vieille gave the query-subquery recursive (QSQR) evaluation method for Datalog deductive databases, which is a top-down method based on tabled SLD-resolution and the set-at-a-time technique. The first version of QSQR [69] is incomplete [43, 71]. As pointed out by Mohamed Yahya [39], the version given in the book [1] is also incomplete. The work [39] corrects and generalizes the QSQR method for Horn knowledge bases. The correction depends on clearing global "input" relations for each iteration of the main loop. The generalized QSQR method for Horn knowledge bases [39] uses the steering control of the corrected QSQR method as in the case of Datalog but does not use adornments and annotations. It uses "input" and "answer" relations consisting of tuples of terms (which may contain variables and function symbols) as well as "supplementary" relations consisting of substitutions.

The QSQ (query-subquery) approach for Datalog queries, as presented in [1], originates from the QSQR method but allows a variety of control strategies. The QSQ framework (including QSQR) for Datalog uses adornments to simulate SLD-resolution in pushing constant symbols from goals to subgoals. The annotated version of QSQ for Datalog uses annotations to simulate SLD-resolution in pushing repeats of variables from goals to subgoals (see [1]).

The magic-sets technique [7, 8] is another formulation of tabling for Datalog deductive databases. It simulates the top-down QSQR evaluation by rewriting the program together with the given query to another equivalent one that when evaluated using a bottom-up technique (e.g., the improved semi-naive evaluation) produces only facts produced by the QSQR evaluation. Thus, it combines the advantages of top-down and bottom-up techniques. Adornments are used as in the QSQR evaluation. To simulate annotations, the magic-sets transformation is augmented with subgoal rectification (see, e.g., [1]). For the connection between top-down and bottom-up approaches to Datalog deductive databases we refer the reader to Bry's work [9]. The Generalized Supplementary Magic Sets algorithm proposed by Beeri and Ramakrishnan [8] uses some special predicates called "supplementary magic predicates" in order to eliminate the duplicate work during the processing. Some authors have extended the magic-sets technique and related ones for Horn knowledge bases [49, 55, 59]. To deal with non-range-restrictedness and function symbols, "magic predicates" are used without adornments [55, 59].

To develop evaluation procedures for Horn knowledge bases one can also adapt tabled SLD-resolution systems of logic programming to reduce the number of accesses to secondary storage. SLD-AL resolution [70, 71] is such a system. In [71], Vieille adapted SLD-AL resolution to Datalog deductive databases to obtain the top-down QoSaQ evaluation method by representing (sets of) goals by means of (sets of) tuples and translating the operations of SLD-AL on goals into operations on tuples. This evaluation method can be implemented as a set-oriented procedure, but Vieille stated that *"We would like, however, to go even further and to claim that the practical interest of our approach lies in its one-inference-at-a-time basis, as opposed to having a set-theoretic basis. First, this tuple-based computational model permits a fine analysis of the duplicate elimination issue. ..."* [71, page 5]. Moreover, the specific techniques of QoSaQ like "instantiation pattern", "rule compilation", "projection" are heavily based on the range-restrictedness and function-free conditions.

Tabled SLD-resolution systems like OLDT [67] and linear tabulated resolution [60, 72] are also efficient computational procedures for logic programming without redundant recomputations, but they are not directly applicable to Horn knowledge bases to obtain efficient evaluation engines because they are not set-oriented (set-at-a-time). In particular, the suspension-resumption mechanism and the stack-wise representation are both tuple-oriented (tuple-at-a-time). Data structures for them are too complex so that they must be dropped if one wants to convert the methods to efficient set-oriented methods. One can use, e.g., XSB [57, 58] (a state-of-the-art implementation of OLDT) as a Horn knowledge base engine, but as pointed out in [28], it is tuple-oriented and not suitable for efficient access to secondary storage. Breadth-First XSB [28] converts XSB to a set-oriented engine [28], but it abandons some essential features of XSB.[1]

Various optimization techniques have been proposed for query processing (see, e.g., [42, 48, 53, 61, 65]). One of them is to reduce the number of materialized intermediate results during the processing by using tail-recursion elimination. In [53], Ross integrated the Magic-Sets evaluation method with a form of tail-recursion elimination. It improves the performance of query evaluation by not materializing the extension of intermediate views.

Positive logic programs can express only monotonic queries. As many queries of practical interest are non-monotonic, it is desirable to consider normal logic programs, which allow negation to occur in the bodies of program clauses. A number of interesting semantics for normal logic programs has been defined, for instance, stratified semantics [2] (for stratified logic programs), stable-model semantics [30] and well-founded semantics [29]. The survey [4] provides a good source for references on these semantics. A normal logic program is stratifiable if it can be divided into strata such that if a negative literal of a predicate p occurs in the body of a program clause in a stratum, then the clauses defining p must belong to an earlier stratum. Programs in this class have a very intuitive semantics and have been considered in [2, 6, 32, 35, 41].

Appendix A contains a more detailed description of some well-known query evaluation methods for Horn knowledge bases.

[1] The original XSB uses depth-first search, while Breadth-First XSB [28] does not.

1.2 Motivation

The most well-known methods for evaluating queries to Datalog deductive databases or Horn knowledge bases are QSQR and Magic-Sets (by Magic-Sets we mean the evaluation method that combines the magic-set transformation with the improved semi-naive bottom-up evaluation method). Both of these methods are goal-directed. As observed by Vieille [71], the QSQR approach is like iterative deepening search. It allows redundant recomputations (see [39, Remark 3.2]). On the other hand, the Magic-Sets method applies breadth-first search. The following example shows that the breadth-first approach is not always efficient.

Example 1.1. The order of program clauses and the order of atoms in the bodies of program clauses may be essential, e.g., when the positive logic program that defines intensional predicates is specified using the Prolog programming style. In such cases, the top-down depth-first approach may be much more efficient than the breadth-first approach. Here is such an example, in which p, q_1 and q_2 are intensional predicates, r_1 and r_2 are extensional predicates, x, y and z are variables, a_i and $b_{i,j}$ are constant symbols:

- the positive logic program:

$$
\begin{array}{l}
p \leftarrow q_1(a_0, a_m) \\
p \leftarrow q_2(a_0, a_m) \\[4pt]
q_1(x, y) \leftarrow r_1(x, y) \\
q_1(x, y) \leftarrow r_1(x, z), q_1(z, y) \\[4pt]
q_2(x, y) \leftarrow r_2(x, y) \\
q_2(x, y) \leftarrow r_2(x, z), q_2(z, y)
\end{array}
$$

- the extensional instance (illustrated in Figure 1.1):

$$
\begin{array}{rcl}
I(r_1) & = & \{(a_i, a_{i+1}) \mid 0 \leq i < m\} \\[4pt]
I(r_2) & = & \{(a_0, b_{1,j}) \mid 1 \leq j \leq n\} \cup \\
& & \{(b_{i,j}, b_{i+1,j}) \mid 1 \leq i < m-1 \text{ and } 1 \leq j \leq n\} \cup \\
& & \{(b_{m-1,j}, a_m) \mid 1 \leq j \leq n\}
\end{array}
$$

- the query: $\leftarrow p$.

Notice that the depth-first approach needs only $\Theta(m)$ steps for evaluating the query, while the breadth-first approach performs $\Theta(m \cdot n)$ steps. When n is comparable to m, the difference is too big. The magic-sets transformation does not help for this case. ■

Our postulate is that the breadth-first approach (including the Magic-Sets evaluation method) is inflexible and not always efficient. Of course, depth-first search is not always good either. The aim of this dissertation is to develop evaluation methods for evaluating queries to Horn knowledge bases that are more efficient than the QSQR evaluation method and more adjustable than the Magic-Sets evaluation method. In particular, good methods should be not only set-oriented and goal-directed but should also reduce computational redundancy as much as possible and allow various control strategies.

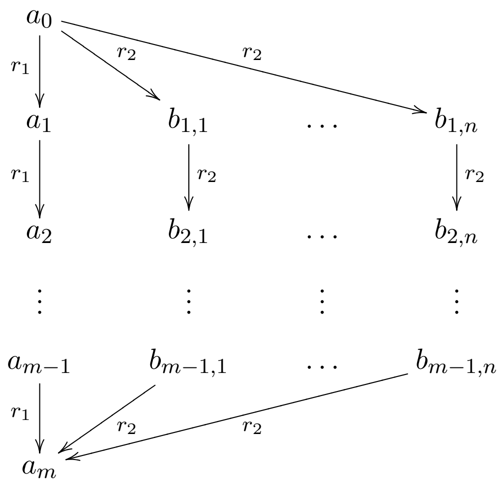


Fig. 1.1: An illustration for the extensional instance given in Example 1.1.

1.3 Contributions

In this dissertation, we make the following main contributions:

- We formulate the query-subquery nets and use them to develop the first framework for developing algorithms for evaluating queries to Horn knowledge bases with the following good properties:
  - the approach is goal-directed,
  - each subquery is processed only once,
  - each supplement tuple, if desired, is transferred only once,
  - operations are done set-at-a-time,
  - any control strategy can be used.

  The intention of our framework is to increase efficiency of query processing by eliminating redundant computation, increasing adjustability[2] and reducing the number of accesses to the secondary storage. The framework forms a generic evaluation method called QSQN. It is sound and complete, and has polynomial time data complexity when the term-depth bound is fixed. The results were published in [45, 46] and presented in Chapter 3.
- We implement QSQN together with the control strategies Disk Access Reduction (DAR) and Depth-First Search (DFS) to obtain the corresponding evaluation methods QSQN-DAR and QSQN-DFS. We also implement the Magic-Sets and QSQR methods for comparison. The comparison is made with respect to the number of read/write operations on relations and the execution time. The results were published in [11].
- We propose a control strategy called Improved Depth-First Control Strategy (IDFS) and implement QSQN together with this strategy to obtain a corresponding evalua-

[2] By "adjustability" we mean easiness in adopting advanced control strategies.

tion method QSQN-IDFS. We came up to the improvement by using query-subquery nets to observe which relations are likely to grow or saturate and which ones are not yet affected by the computation and the other relations. Our intention is to accumulate as many as possible tuples or subqueries at each node of the query-subquery net before processing it. The details are described in Section 6.1. The comparison between QSQN-IDFS and QSQN-DFS with respect to the number of read/write operations on relations was published in [16].

- We make a comparison between the implemented QSQN-IDFS, QSQR and Magic-Sets methods using representative examples that appeared in well-known articles on deductive databases as well as new examples. The results are shown in Chapter 6. The comparison is made with respect to the following measures:
  - the number of read or write operations on relations,
  - the maximum number of tuples and subqueries kept in the computer memory,
  - the number of accesses to the secondary storage as well as the number of tuples and subqueries read from or written to the secondary storage when the memory is limited.
- We incorporate tail-recursion elimination into query-subquery nets in order to obtain the QSQN-TRE evaluation method for Horn knowledge bases. The aim is to reduce materializing the intermediate results during the processing of a query with tail-recursion. We prove the soundness and completeness of the proposed evaluation method and show that, when the term-depth bound is fixed, the QSQN-TRE method has polynomial time data complexity. We specify the QSQN-TRE method in detail in Section 4.1. The results were published in [17].
- We extend QSQN-TRE to obtain an evaluation method called QSQN-rTRE, which can eliminate not only tail-recursive predicates but also intensional predicates that appear rightmost in the bodies of the program clauses. The aim is to reduce materializing the intermediate results (when desired) during the processing. The method was published in [14] and is presented in Section 4.2.
- We incorporate stratified negation into query-subquery nets to obtain a method called QSQN-STR for evaluating queries to stratified knowledge bases. The proposed method was published in [15] and is discussed in Chapter 5.

This dissertation was written by me, having important comments and suggestions from my supervisors, dr hab. Linh Anh Nguyen and dr. Joanna Golińska-Pilarek. Regarding the published works mentioned in this dissertation, the first one [46] is an ICCCI'2012 conference paper, whose long version is the manuscript [45]. In the works [45, 46], Nguyen and I discussed the scientific problems and solutions associated with the study. These papers were written mainly by Nguyen and presented by me at the ICCCI'2012 conference. I myself wrote all the remaining published works [11, 14, 15, 16, 17] mentioned in this dissertation and presented them at the corresponding international conferences. For these publications, I received a lot of useful technical comments and suggestions from my supervisors. They also corrected the English grammar for the drafts of my published papers as well as for this dissertation. I myself also implemented all of the mentioned methods in Java for the comparison between them and provided all of the experimental results.

1.4 The Structure of This Dissertation

The rest of the dissertation is organized as follows:

Chapter 2: This chapter recalls the notions and definitions of first-order logic that are related to the topic of this dissertation.

Chapter 3: In this chapter, we formulate the query-subquery nets framework for developing algorithms for evaluating queries to Horn knowledge bases. The framework forms a generic evaluation method called QSQN. We present an illustrative example, a pseudocode and properties of the evaluation algorithm.

Chapter 4: In the first section of this chapter, we present the QSQN-TRE method for evaluating queries to Horn knowledge bases by incorporating tail-recursion elimination into query-subquery nets. We give an intuition and a formal definition of such modified nets as well as explanations, an illustrative example and a pseudocode of the evaluation algorithm. Furthermore, we prove the soundness and completeness of the QSQN-TRE method. Then, we extend the QSQN-TRE method to obtain another method called QSQN-rTRE in the next section.

Chapter 5: In this chapter, we present the QSQN-STR evaluation method for evaluating queries to stratified knowledge bases. Additionally, we prove the soundness and completeness of QSQN-STR for the case without function symbols.

Chapter 6: In this chapter, we first present the IDFS control strategy, which can be used for QSQN, QSQN-TRE and QSQN-rTRE. We then provide the experimental results and a discussion on the performance of the proposed evaluation methods. In order to compare our methods with the well-known evaluation methods such that QSQR and Magic-Sets, we have implemented all of these methods. We compare them using representative examples that appear in many articles on deductive databases as well as new ones. We also report experimental results of QSQN-STR using a control strategy called IDFS2, which is a modified version of IDFS.

Chapter 7: The final chapter draws some conclusions and indicates directions for future work.

This dissertation includes five appendices: Appendix A discusses the well-known methods QSQR and Magic-Sets for evaluating queries to Horn knowledge bases together with their pros and cons. Appendix B contains a part of the proof of the completeness of QSQN-TRE. Appendices C, D and E contain functions and procedures used for QSQN-TRE, QSQN-rTRE and QSQN-STR, respectively. In addition, the bibliography, the lists of figures and tables as well as an index of symbols and terms are provided at the end of this dissertation.

Chapter 2

Preliminaries

This chapter recalls the classical notions and definitions from first-order logic and database theory which can be found, e.g., in [1, 37]. Most of our exposition here is taken from Section 2 of [39], with minor modifications.

Definition 2.1. A *signature* for first-order logic is a tuple $\Sigma = \langle \mathcal{V}, \mathcal{C}, \mathcal{F}, \mathcal{P} \rangle$ consisting of the following pairwise disjoint sets:

- a finite set $\mathcal{V}$ of variable symbols,
- a finite set $\mathcal{C}$ of constant symbols,
- a finite set $\mathcal{F}$ of function symbols,
- a finite set $\mathcal{P}$ of predicates (also called relation symbols). ■

The following notions are defined over a fixed signature, thus we shall use $\Sigma = \langle \mathcal{V}, \mathcal{C}, \mathcal{F}, \mathcal{P} \rangle$ without mentioning it further. Terms and formulas over a fixed signature are defined in the usual way as follows.

Definition 2.2 (Term). A *term* is defined inductively as follows:

- A variable is a term.
- A constant is a term.
- If f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term. ■

Definition 2.3 (Formula). A *formula* is defined inductively as follows:

- If p is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is a formula (called an *atomic formula* or *atom* for short).
- If φ and ψ are formulas, then so are $(\neg\varphi)$, $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \rightarrow \psi)$ and $(\varphi \leftrightarrow \psi)$.
- If φ is a formula and x is a variable, then $(\forall x\, \varphi)$ and $(\exists x\, \varphi)$ are formulas. ■

Definition 2.4 (Literal). A *literal* is an atom or the negation of an atom. A *positive literal* is an atom. A *negative literal* is the negation $\neg\varphi$ of an atom φ. ■

Definition 2.5 (Expression). An *expression* is either a term, a tuple of terms, a formula without quantifiers or a list of formulas without quantifiers. A *simple expression* is either a term or an atom. ■

The *term-depth of an expression* is the maximal nesting depth of function symbols occurring in that expression.

Definition 2.6 (Ground Term/Atom/Literal). A *ground term* is a term without variables. A *ground atom* is an atom with ground terms as its arguments. A *ground literal* is a literal constructed from a ground atom. ■

Definition 2.7 (Interpretation/Variable Assignment). An *interpretation* is a pair $\mathcal{I} = \langle \mathcal{D}, \cdot^{\mathcal{I}} \rangle$ consisting of

– a nonempty set $\mathcal{D}$ called the *domain* (or *universe*), and
– a function $\cdot^{\mathcal{I}}$ that assigns a meaning to constant, function and predicate symbols:
  - $c^{\mathcal{I}} \in \mathcal{D}$ for each constant symbol $c \in \mathcal{C}$,
  - $f^{\mathcal{I}} : \mathcal{D}^n \to \mathcal{D}$ for each n-ary function symbol $f \in \mathcal{F}$,
  - $p^{\mathcal{I}} \subseteq \mathcal{D}^n$ for each n-ary predicate $p \in \mathcal{P}$.

A *variable assignment* is a function α that maps variables to elements in the domain $\mathcal{D}$, i.e., $\alpha : \mathcal{V} \to \mathcal{D}$. ■

Definition 2.8 (Interpretation of a Term). Let $\mathcal{I} = \langle \mathcal{D}, \cdot^{\mathcal{I}} \rangle$ be an interpretation, α a variable assignment, and t a term. The interpretation of t under $\mathcal{I}$ and α is an element of the domain $\mathcal{D}$ defined as follows:

– if $t = x$ then $x^{\mathcal{I},\alpha} = \alpha(x)$,
– if $t = c$ then $c^{\mathcal{I},\alpha} = c^{\mathcal{I}}$,
– if $t = f(t_1, \dots, t_n)$ then $(f(t_1, \dots, t_n))^{\mathcal{I},\alpha} = f^{\mathcal{I}}(t_1^{\mathcal{I},\alpha}, \dots, t_n^{\mathcal{I},\alpha})$. ■

Definition 2.9 (Satisfaction Relation). Let $\mathcal{I} = \langle \mathcal{D}, \cdot^{\mathcal{I}} \rangle$ be an interpretation, α a variable assignment, Γ a set of formulas, φ, ψ formulas, and $p(t_1, \dots, t_n)$ an atom. Then

$$
\begin{array}{lll}
\mathcal{I}, \alpha \models p(t_1, \dots, t_n) & \text{iff} & (t_1^{\mathcal{I},\alpha}, \dots, t_n^{\mathcal{I},\alpha}) \in p^{\mathcal{I}} \\
\mathcal{I}, \alpha \models \neg p(t_1, \dots, t_n) & \text{iff} & (t_1^{\mathcal{I},\alpha}, \dots, t_n^{\mathcal{I},\alpha}) \notin p^{\mathcal{I}} \\
\mathcal{I}, \alpha \models \varphi \wedge \psi & \text{iff} & \mathcal{I}, \alpha \models \varphi \text{ and } \mathcal{I}, \alpha \models \psi \\
\mathcal{I}, \alpha \models \varphi \vee \psi & \text{iff} & \mathcal{I}, \alpha \models \varphi \text{ or } \mathcal{I}, \alpha \models \psi \\
\mathcal{I}, \alpha \models \forall x\, \varphi & \text{iff} & \mathcal{I}, \alpha_d^x \models \varphi \text{ for all } d \in \mathcal{D} \\
\mathcal{I}, \alpha \models \exists x\, \varphi & \text{iff} & \mathcal{I}, \alpha_d^x \models \varphi \text{ for at least one } d \in \mathcal{D}
\end{array}
$$

where α_d^x is the variable assignment such that, for every $y \in \mathcal{V}$:

$$
\alpha_d^x(y) = \begin{cases} d & \text{if } y \text{ is } x, \\ \alpha(y) & \text{otherwise.} \end{cases}
$$

The binary satisfaction relation $\models$ between an interpretation $\mathcal{I}$ and a formula φ (or a set of formulas Γ) is defined as follows:

$$
\begin{array}{lll}
\mathcal{I} \models \varphi & \text{iff} & \mathcal{I}, \alpha \models \varphi \text{ for all assignments } \alpha : \mathcal{V} \to \mathcal{D}, \\
\mathcal{I} \models \Gamma & \text{iff} & \mathcal{I} \models \varphi \text{ for all } \varphi \in \Gamma.
\end{array}
$$

If $\mathcal{I} \models \varphi$ then we say that $\mathcal{I}$ *satisfies* φ (or φ is *true* in $\mathcal{I}$). If $\mathcal{I} \models \varphi$ (resp. $\mathcal{I} \models \Gamma$) then $\mathcal{I}$ is a *model* of φ (resp. Γ). If φ (resp. Γ) has a model then it is *satisfiable*, otherwise it is *unsatisfiable*. If $\mathcal{I} \models \Gamma$ implies $\mathcal{I} \models \varphi$ for all interpretations $\mathcal{I}$, then φ is a *logical consequence* of Γ, denoted by $\Gamma \models \varphi$. ■

2.1 Substitution and Unification

Definition 2.10 (Substitution). A *substitution* is a finite set $\theta = \{x_1/t_1, \dots, x_k/t_k\}$, where $x_1, \dots, x_k$ are pairwise distinct variables, $t_1, \dots, t_k$ are terms, and $t_i \neq x_i$ for all $1 \leq i \leq k$. The *empty substitution* is denoted by ε. ∎

In what follows, the set $dom(\theta) = \{x_1, \dots, x_k\}$ is called the *domain* of θ, the set $range(\theta) = \{t_1, \dots, t_k\}$ is called the *range* of θ. The *restriction of a substitution* θ to a set X of variables is the substitution $\theta_{|X} = \{(x/t) \in \theta \mid x \in X\}$. The *term-depth of a substitution* is the maximal nesting depth of function symbols occurring in that substitution.

Let $\theta = \{x_1/t_1, \dots, x_k/t_k\}$ be a substitution and E be an expression. Then $E\theta$, the *instance* of E by θ, is the expression obtained from E by simultaneously replacing all occurrences of the variable x_i in E by the term t_i, for $1 \leq i \leq k$.

Let $\theta = \{x_1/t_1, \dots, x_k/t_k\}$ and $\delta = \{y_1/s_1, \dots, y_h/s_h\}$ be substitutions (where $x_1, \dots, x_k$ are pairwise distinct variables, and $y_1, \dots, y_h$ are also pairwise distinct variables). Then the *composition* $\theta\delta$ of θ and δ is the substitution obtained from the sequence $\{x_1/(t_1\delta), \dots, x_k/(t_k\delta), y_1/s_1, \dots, y_h/s_h\}$ by deleting any binding $x_i/(t_i\delta)$ for which $x_i = (t_i\delta)$ and deleting any binding y_j/s_j for which $y_j \in \{x_1, \dots, x_k\}$.

A substitution θ is *idempotent* if $\theta\theta = \theta$. It is known that $\theta = \{x_1/t_1, \dots, x_k/t_k\}$ is idempotent if none of $x_1, \dots, x_k$ occurs in any $t_1, \dots, t_k$.

If θ and δ are substitutions such that $\theta\delta = \delta\theta = \varepsilon$, then we call them *renaming substitutions*. We say that an expression E is a *variant* of an expression E' if there exist substitutions θ and γ such that $E = E'\theta$ and $E' = E\gamma$.

Definition 2.11 (Generality of Substitutions). A substitution θ is *more general* than a substitution δ if there exists a substitution γ such that $\delta = \theta\gamma$. ∎

Note that, according to this definition, θ is more general than itself.

Definition 2.12 (Unifier). Let Γ be a set of simple expressions. A substitution θ is called a *unifier* for Γ if $\Gamma\theta$ is a singleton. If $\Gamma\theta = \{\varphi\}$ then we say that θ unifies Γ (into φ). ∎

Definition 2.13 (Most General Unifier). A unifier θ for Γ is called a *most general unifier* (mgu) for Γ if θ is more general than every unifier of Γ. ∎

There is an effective algorithm, called the *unification algorithm*, for checking whether a set Γ of simple expressions is unifiable (i.e., has a unifier) and computing an idempotent mgu for Γ if Γ is unifiable (see, e.g., [37]).

If E is an expression or a substitution then by $Vars(E)$ we denote the set of variables occurring in E. If φ is a formula then by $\forall(\varphi)$ we denote the *universal closure* of φ, which is the formula obtained by adding a universal quantifier for every variable having a free occurrence in φ.

2.2 Positive Logic Programs and SLD-Resolution

Definition 2.14 (Positive Program Clause). A *positive* (or *definite*) *program clause* is a formula of the form $\forall(A \vee \neg B_1 \vee \ldots \vee \neg B_k)$ with $k \geq 0$, written as $A \leftarrow B_1, \ldots, B_k$, where A, B_1, ..., B_k are atoms. A is called the *head*, and $(B_1, \ldots, B_k)$ the *body* of the program clause. If $k = 0$ then the clause is called a *unit clause* with the form $A \leftarrow$, (i.e., a definite program clause with an empty body). If p is the predicate of A then the program clause is called a program clause defining p. ■

Definition 2.15 (Positive Logic Program). A *positive* (or *definite*) *logic program* is a finite set of (positive) program clauses. ■

Definition 2.16 (Goal). A *goal* (also called a *negative clause*) is a formula of the form $\forall(\neg B_1 \vee \ldots \vee \neg B_k)$, written as $\leftarrow B_1, \ldots, B_k$, where $B_1, \ldots, B_k$ are atoms. If $k = 1$ then the goal is called a *unary goal*. If $k = 0$ then the goal stands for falsity and is called the *empty goal* (or the *empty clause*) and denoted by $\Box$. ■

Definition 2.17 (Correct Answer). If P is a positive logic program and $G = \leftarrow B_1, \ldots, B_k$ is a goal, then θ is called a *correct answer* for $P \cup \{G\}$ if $P \models \forall((B_1 \wedge \ldots \wedge B_k)\theta)$. ■

We now give definitions for SLD-resolution.

Definition 2.18 (SLD-Resolvent). A goal G' is *derived* from a goal $G = \leftarrow A_1, \ldots, A_i, \ldots, A_k$ and a program clause $\varphi = (A \leftarrow B_1, \ldots, B_h)$ using A_i as the *selected atom* and θ as the most general unifier (mgu) if θ is an mgu for A_i and A, and $G' = \leftarrow (A_1, \ldots, A_{i-1}, B_1, \ldots, B_h, A_{i+1}, \ldots, A_k)\theta$. We call G' a *resolvent* of G and φ. If $i = 1$ then we say that G' is derived from G and φ using *the leftmost selection function*. ■

Let P be a positive logic program and G be a goal.

Definition 2.19 (SLD-Derivation). An *SLD-derivation* from $P \cup \{G\}$ consists of a (finite or infinite) sequence $G_0 = G$, G_1, G_2, ...of goals, a sequence $\varphi_1, \varphi_2, \ldots$ of variants of program clauses of P and a sequence $\theta_1, \theta_2, \ldots$ of mgu's such that each G_{i+1} is derived from G_i and φ_{i+1} using θ_{i+1}. ■

Note that, each φ_i is a suitable variant of the corresponding program clause. That is, φ_i does not have any variables which already appear in the derivation up to G_{i-1}. Each program clause variant φ_i is called an *input program clause*.

Definition 2.20 (SLD-Refutation). An *SLD-refutation* of $P \cup \{G\}$ is a finite SLD-derivation from $P \cup \{G\}$ which has the empty clause as the last goal in the derivation. ■

Definition 2.21 (Computed Answer). A *computed answer* θ for $P \cup \{G\}$ is the substitution obtained by restricting the composition $\theta_1 \ldots \theta_n$ to the variables of G, where $\theta_1, \ldots, \theta_n$ is the sequence of mgu's occurring in an SLD-refutation of $P \cup \{G\}$. ■

Theorem 2.1 (Soundness and Completeness of SLD-Resolution [24, 63]). *Let P be a positive logic program and G be a goal. Then every computed answer for $P \cup \{G\}$ is a correct answer for $P \cup \{G\}$. Conversely, for every correct answer θ for $P \cup \{G\}$, using any selection function there exists a computed answer δ for $P \cup \{G\}$ such that $G\theta = G\delta\gamma$ for some substitution γ.* ■

We will use also the following well-known lemma:

Lemma 2.2 (Lifting Lemma). *Let P be a positive logic program, G be a goal, θ be a substitution, and l be a natural number. Suppose there exists an SLD-refutation of $P \cup \{G\theta\}$ using mgu's $\theta_1, \ldots, \theta_n$ such that the variables of the input program clauses are distinct from the variables in G and θ and the term-depths of the goals are bounded by l. Then there exist a substitution γ and an SLD-refutation of $P \cup \{G\}$ using the same sequence of input program clauses, the same selected atoms and mgu's $\theta'_1, \ldots, \theta'_n$ such that the term-depths of the goals are bounded by l and $\theta\theta_1 \ldots \theta_n = \theta'_1 \ldots \theta'_n\gamma$.* ■

The Lifting Lemma given in [37] does not contain the condition "the variables of the input program clauses are distinct from the variables in G and θ" and is therefore inaccurate (see, e.g., [3]). The correct version given above follows from the one presented, amongst others, in [62]. For applications of this lemma in this paper, we assume that *fresh variables* from a special infinite list of variables are used for renaming variables of input program clauses in SLD-derivations, and that mgu's are computed using a standard method. The mentioned condition will thus be satisfied.

In a computational process, a *fresh variant* of a formula φ, where φ can be an atom, a goal $\leftarrow A$ or a program clause $A \leftarrow B_1, \ldots, B_k$ (written without quantifiers), is a formula $\varphi\theta$, where θ is a renaming substitution such that $dom(\theta) = Vars(\varphi)$ and $range(\theta)$ consists of fresh variables that were not used in the computation (and the input).

2.3 Definitions for Horn Knowledge Bases

Similarly as for deductive databases, we classify each predicate either as *intensional* or as *extensional*. A *generalized tuple* is a tuple of terms, which may contain function symbols and variables. A *generalized relation* is a set of generalized tuples of the same arity.

Definition 2.22 (Horn Knowledge Base). A *Horn knowledge base* is defined to be a pair (P, I), where P is a positive logic program for defining intensional predicates, and I is a *generalized extensional instance*, which is a mapping that associates each extensional n-ary predicate with an n-ary generalized relation. ■

Note that intensional predicates are defined by a positive logic program which may contain function symbols and not be range-restricted. From now on, we use the term "relation" to mean a generalized relation, and the term "extensional instance" to mean a generalized extensional instance.

Note also that, we will treat a tuple $\bar{t}$ from a relation associated with a predicate p as the atom $p(\bar{t})$. Thus, a relation (of tuples) of a predicate p is a set of atoms of p, and an extensional instance is a set of atoms of extensional predicates. Conversely, a set of atoms of p can be treated as a relation (of tuples) of the predicate p.

Given a Horn knowledge base specified by a positive logic program P and an extensional instance I, a *query* to the knowledge base is a positive formula $\varphi(\overline{x})$ without quantifiers, where $\overline{x}$ is a tuple of all the variables of φ.[1] A (*correct*) *answer* for the query is a tuple $\overline{t}$ of terms of the same length as $\overline{x}$ such that $P \cup I \models \forall(\varphi(\overline{t}))$. When measuring *data complexity*, we assume that P and φ are fixed, while I varies. Thus, the pair $(P, \varphi(\overline{x}))$ is treated as a *query* to the extensional instance I. We will use the term "query" in this meaning.

It can be shown that every query $(P, \varphi(\overline{x}))$ can be transformed in polynomial time to an equivalent query of the form $(P', q(\overline{x}))$ over a signature extended with new intensional predicates, including q. The equivalence means that, for every extensional instance I and every tuple $\overline{t}$ of terms of the same length as $\overline{x}$, $P \cup I \models \forall(\varphi(\overline{t}))$ iff $P' \cup I \models \forall(q(\overline{t}))$. The transformation is based on introducing new predicates for defining complex subformulas occurring in the query. For example, if $\varphi = p(x) \wedge r(x, y)$, then $P' = P \cup \{q(x, y) \leftarrow p(x), r(x, y)\}$, where q is a new intensional predicate.

Without loss of generality, we will consider only queries of the form $(P, q(\overline{x}))$, where q is an intensional predicate. Answering such a query on an extensional instance I is to find (correct) answers for $P \cup I \cup \{\leftarrow q(\overline{x})\}$.

Definition 2.23. We say that a predicate p *directly depends* on a predicate q if the considered program P has a clause defining p that uses q in the body. We define the relation *"depends"* to be the reflexive and transitive closure of "directly depends". ■

[1] A *positive formula without quantifiers* is a formula built up from atoms using only connectives $\wedge$ and $\vee$.

Chapter 3

The Query-Subquery Net Evaluation Method

In this chapter, we generalize the QSQ approach for Horn knowledge bases. Given a positive logic program, we make a query-subquery net structure and use it as a flow control network to determine which subqueries in which nodes should be processed next. We show how the data are transferred through edges of the net. We also propose an algorithm together with related procedures and functions for this framework. The algorithm repeatedly selects an active edge and fires the operation for the edge to transfer unprocessed data. Such a selection is decided by the adopted control strategy, which can be arbitrary. In addition, the processing is divided into smaller steps which can be delayed to maximize adjustability and allow various control strategies. The intention is to increase efficiency of query processing by eliminating redundant computation, increasing adjustability and reducing the number of accesses to the secondary storage. From now on, by a "program" we mean a positive logic program.

This chapter is organized as follows. Section 3.1 presents definitions and examples of the query-subquery net evaluation method for Horn knowledge bases. Section 3.2 presents an algorithm together with its properties. The preliminary experiments and a discussion on the performance of the proposed method are provided later in Chapter 6.

3.1 Query-Subquery Nets

In what follows, P is a positive logic program and $\varphi_1, \ldots, \varphi_m$ are all the program clauses of P, with $\varphi_i = (A_i \leftarrow B_{i,1}, \ldots, B_{i,n_i})$, for $1 \leq i \leq m$ and $n_i \geq 0$. The following definition shows how to make a QSQ-net structure from the given logic program P.

Definition 3.1 (Query-Subquery Net Structure). A *query-subquery net structure* (*QSQ-net structure* for short) of P is a tuple (V, E, T) such that:

- V is a set of nodes that consists of:
  - *input_p* and *ans_p*, for each intensional predicate p of P,
  - *pre_filter*$_i$, *filter*$_{i,1}$, ..., *filter*$_{i,n_i}$, *post_filter*$_i$, for each $1 \leq i \leq m$.
- E is a set of edges that consists of:

- $(\mathit{filter}_{i,1}, \mathit{filter}_{i,2})$, ..., $(\mathit{filter}_{i,n_i-1}, \mathit{filter}_{i,n_i})$, for each $1 \leq i \leq m$,
- $(\mathit{pre\_filter}_i, \mathit{filter}_{i,1})$ and $(\mathit{filter}_{i,n_i}, \mathit{post\_filter}_i)$, for each $1 \leq i \leq m$ with $n_i \geq 1$,
- $(\mathit{pre\_filter}_i, \mathit{post\_filter}_i)$, for each $1 \leq i \leq m$ with $n_i = 0$,
- $(\mathit{input\_p}, \mathit{pre\_filter}_i)$ and $(\mathit{post\_filter}_i, \mathit{ans\_p})$, for each $1 \leq i \leq m$, where p is the predicate of A_i,
- $(\mathit{filter}_{i,j}, \mathit{input\_p})$ and $(\mathit{ans\_p}, \mathit{filter}_{i,j})$, for each intensional predicate p and each $1 \leq i \leq m$ and $1 \leq j \leq n_i$ such that $B_{i,j}$ is an atom of p.

– T is a function, called the *memorizing type* of the net structure, mapping each node $\mathit{filter}_{i,j} \in V$ such that the predicate of $B_{i,j}$ is extensional to *true* or *false*. If $T(\mathit{filter}_{i,j}) = \mathit{false}$ (and the predicate of $B_{i,j}$ is extensional) then subqueries for $\mathit{filter}_{i,j}$ are always processed immediately, without being accumulated at $\mathit{filter}_{i,j}$.

If $(v, w) \in E$ then we call w a *successor* of v, and v a *predecessor* of w. Note that V and E are uniquely specified by P. We call the pair (V, E) the *QSQ topological structure* of P. ■

Example 3.1. Consider the following (recursive) positive logic program, where x, y and z are variables, p is an intensional predicate, and q is an extensional predicate:

$$p(x, y) \leftarrow q(x, y)$$
$$p(x, y) \leftarrow q(x, z), p(z, y).$$

Its QSQ topological structure is illustrated in Figure 3.1. ■

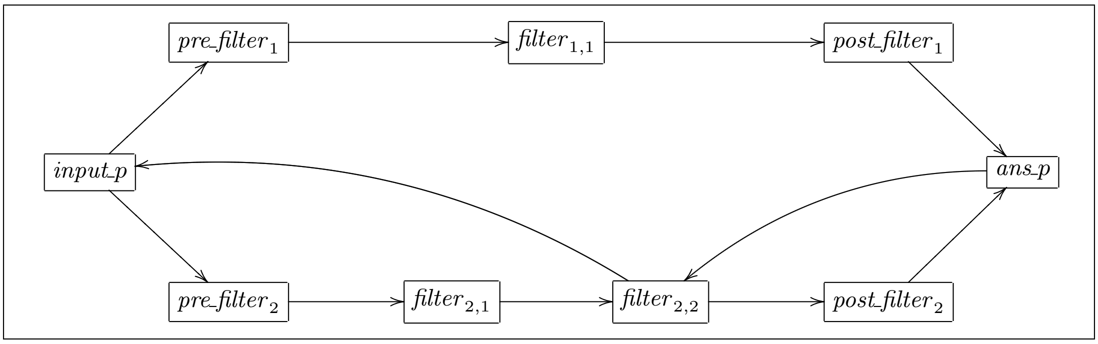


Fig. 3.1: The QSQ topological structure of the program given in Example 3.1.

Example 3.2. Consider the following (non-recursive) logic program, where x, y and z are variables, p and r are intensional predicates, q, s and t are extensional predicates:

$$p(x, y) \leftarrow q(x, z), r(z, y)$$
$$r(x, y) \leftarrow s(x, y)$$
$$r(x, y) \leftarrow t(x, y).$$

This program is a modified version of an example from [72]. Figure 3.2 illustrates the QSQ topological structure of this program. ■

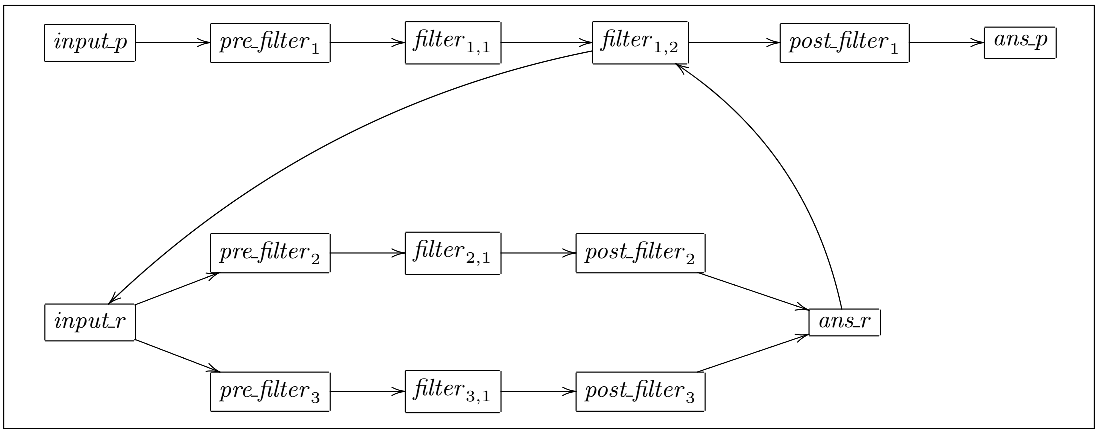


Fig. 3.2: The QSQ topological structure of the program given in Example 3.2.

Definition 3.2 (Query-Subquery Net). A *query-subquery net* (*QSQ-net* for short) of P is a tuple $N = (V, E, T, C)$ such that (V, E, T) is a QSQ-net structure of P, C is a mapping that associates each node $v \in V$ with a structure called the *contents* of v, and the following conditions are satisfied:

- $C(v)$, where $v = \mathit{input\_p}$ or $v = \mathit{ans\_p}$ for an intensional predicate p of P, consists of:
  - $\mathit{tuples}(v):$ a set of generalized tuples of the same arity as p,
  - $\mathit{unprocessed}(v, w)$ for each $(v, w) \in E$: a subset of $\mathit{tuples}(v)$.
- $C(v)$, where $v = \mathit{pre\_filter}_i$, consists of:
  - $\mathit{atom}(v) = A_i$ and $\mathit{post\_vars}(v) = \mathit{Vars}((B_{i,1}, \ldots, B_{i,n_i}))$,
- $C(v)$, where $v = \mathit{post\_filter}_i$, is empty, but we assume $\mathit{pre\_vars}(v) = \emptyset$.
- $C(v)$, where $v = \mathit{filter}_{i,j}$ and p is the predicate of $B_{i,j}$, consists of:
  - $\mathit{kind}(v) = \mathit{extensional}$ if p is extensional, and
    $\mathit{kind}(v) = \mathit{intensional}$ otherwise,
  - $\mathit{pred}(v) = p$ and $\mathit{atom}(v) = B_{i,j}$,
  - $\mathit{pre\_vars}(v) = \mathit{Vars}((B_{i,j}, \ldots, B_{i,n_i}))$ and
    $\mathit{post\_vars}(v) = \mathit{Vars}((B_{i,j+1}, \ldots, B_{i,n_i}))$,
  - $\mathit{subqueries}(v)$: a set of pairs of the form $(\bar{t}, \delta)$, where $\bar{t}$ is a generalized tuple of the same arity as the predicate of A_i and δ is an idempotent substitution such that $\mathit{dom}(\delta) \subseteq \mathit{pre\_vars}(v)$ and $\mathit{dom}(\delta) \cap \mathit{Vars}(\bar{t}) = \emptyset$,
  - $\mathit{unprocessed\_subqueries}(v) \subseteq \mathit{subqueries}(v)$,
  - in the case p is intensional:
    * $\mathit{unprocessed\_subqueries}_2(v) \subseteq \mathit{subqueries}(v)$,
    * $\mathit{unprocessed\_tuples}(v):$ a set of generalized tuples of the same arity as p.
- if $v = \mathit{filter}_{i,j}$, $\mathit{kind}(v) = \mathit{extensional}$ and $T(v) = \mathit{false}$ then $\mathit{subqueries}(v) = \emptyset$. ■

Figure 3.3 illustrates a QSQ-net of the positive logic program given in Example 3.1.

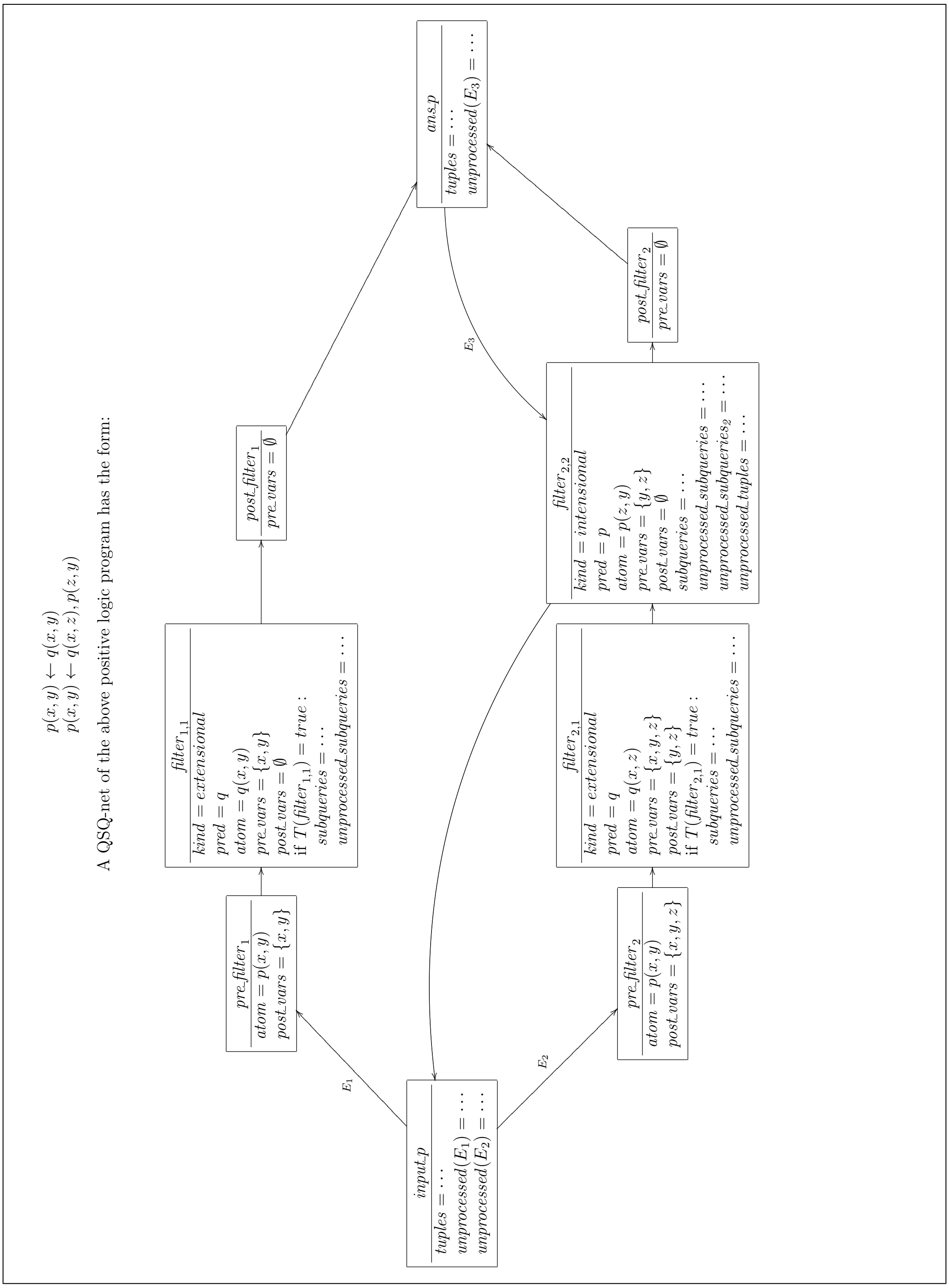


Fig. 3.3: The QSQ-net of the program given in Example 3.1.

By a *subquery* we mean a pair of the form $(\overline{t}, \delta)$, where $\overline{t}$ is a generalized tuple and δ is an idempotent substitution such that $dom(\delta) \cap \mathit{Vars}(\overline{t}) = \emptyset$.

For $v = \mathit{filter}_{i,j}$ and p being the predicate of A_i, the meaning of a subquery $(\overline{t}, \delta) \in \mathit{subqueries}(v)$ is that: for processing a goal $\leftarrow p(\overline{s})$ with $\overline{s} \in \mathit{tuples}(\mathit{input\_p})$ using the program clause $\varphi_i = (A_i \leftarrow B_{i,1}, \ldots, B_{i,n_i})$, unification of $p(\overline{s})$ and A_i as well as processing of the subgoals $B_{i,1}, \ldots, B_{i,j-1}$ were done, amongst others, by using a sequence of mgu's $\gamma_0, \ldots, \gamma_{j-1}$ with the property that $\overline{t} = \overline{s}\gamma_0 \ldots \gamma_{j-1}$ and $\delta = (\gamma_0 \ldots \gamma_{j-1})_{|\mathit{Vars}((B_{i,j},\ldots,B_{i,n_i}))}$.

An *empty QSQ-net of* P is a QSQ-net of P such that all the sets of the form $\mathit{tuples}(v)$, $\mathit{unprocessed}(v,w)$, $\mathit{subqueries}(v)$, $\mathit{unprocessed\_subqueries}(v)$, $\mathit{unprocessed\_subqueries_2}(v)$ or $\mathit{unprocessed\_tuples}(v)$ are empty.

In a QSQ-net, if $v = \mathit{pre\_filter}_i$ or $v = \mathit{post\_filter}_i$ or ($v = \mathit{filter}_{i,j}$ and $\mathit{kind}(v) = \mathit{extensional}$) then v has exactly one successor, which we denote by $\mathit{succ}(v)$.

If v is $\mathit{filter}_{i,j}$ with $\mathit{kind}(v) = \mathit{intensional}$ and $\mathit{pred}(v) = p$ then v has exactly two successors. In that case, let

$$\mathit{succ}(v) = \begin{cases} \mathit{filter}_{i,j+1} & \text{if } n_i > j, \\ \mathit{post\_filter}_i & \text{otherwise,} \end{cases}$$

and $\mathit{succ}_2(v) = \mathit{input\_p}$. The set $\mathit{unprocessed\_subqueries}(v)$ is used for (i.e., corresponds to) the edge $(v, \mathit{succ}(v))$, while $\mathit{unprocessed\_subqueries_2}(v)$ is used for the edge $(v, \mathit{succ}_2(v))$.

Note that if $\mathit{succ}(v) = w$ then $\mathit{post\_vars}(v) = \mathit{pre\_vars}(w)$. In particular, $\mathit{post\_vars}(\mathit{filter}_{i,n_i}) = \mathit{pre\_vars}(\mathit{post\_filter}_i) = \emptyset$.

The formats of data transferred through edges of a QSQ-net are specified as follows:

- data transferred through an edge of the form $(\mathit{input\_p}, v)$, $(v, \mathit{input\_p})$, $(v, \mathit{ans\_p})$ or $(\mathit{ans\_p}, v)$ is a finite set of generalized tuples of the same arity as p,
- data transferred through an edge (u, v) with $v = \mathit{filter}_{i,j}$ and u not being of the form $\mathit{ans\_p}$ is a finite set of subqueries that can be added to $\mathit{subqueries}(v)$,
- data transferred through an edge $(v, \mathit{post\_filter}_i)$ is a set of subqueries $(\overline{t}, \varepsilon)$ such that $\overline{t}$ is a generalized tuple of the same arity as the predicate of A_i.

If $(\overline{t}, \delta)$ and $(\overline{t}', \delta')$ are subqueries that can be transferred through an edge to v then we say that $(\overline{t}, \delta)$ is *more general* than $(\overline{t}', \delta')$ w.r.t. v, and that $(\overline{t}', \delta')$ is *less general* than $(\overline{t}, \delta)$ w.r.t. v, if there exists a substitution γ such that $\overline{t}\gamma = \overline{t}'$ and $(\delta\gamma)_{|\mathit{pre\_vars}(v)} = \delta'$.

Informally, a subquery $(\overline{t}, \delta)$ transferred through an edge to v is processed as follows:

- if $v = \mathit{filter}_{i,j}$, $\mathit{kind}(v) = \mathit{extensional}$ and $\mathit{pred}(v) = p$ then, for each $\overline{t}' \in I(p)$, if $\mathit{atom}(v)\delta = B_{i,j}\delta$ is unifiable with a fresh variant of $p(\overline{t}')$ by an mgu γ then transfer the subquery $(\overline{t}\gamma, (\delta\gamma)_{|\mathit{post\_vars}(v)})$ through $(v, \mathit{succ}(v))$,
- if $v = \mathit{filter}_{i,j}$, $\mathit{kind}(v) = \mathit{intensional}$ and $\mathit{pred}(v) = p$ then
  - transfer the tuple $\overline{t}'$ such that $p(\overline{t}') = \mathit{atom}(v)\delta = B_{i,j}\delta$ through $(v, \mathit{input\_p})$ to add a fresh variant of it to $\mathit{tuples}(\mathit{input\_p})$,
  - for each currently existing $\overline{t}' \in \mathit{tuples}(\mathit{ans\_p})$, if $\mathit{atom}(v)\delta = B_{i,j}\delta$ is unifiable with a fresh variant of $p(\overline{t}')$ by an mgu γ then transfer the subquery $(\overline{t}\gamma, (\delta\gamma)_{|\mathit{post\_vars}(v)})$ through $(v, \mathit{succ}(v))$,

Algorithm 1: for evaluating a query $(P, q(\overline{x}))$ on an extensional instance I.

let (V, E, T) be a QSQ-net structure of P; `// T can be chosen arbitrarily`
set C so that $N = (V, E, T, C)$ is an empty QSQ-net of P;
let $\overline{x}'$ be a fresh variant of $\overline{x}$;
$tuples(input\_q) := \{\overline{x}'\}$;
foreach $(input\_q, v) \in E$ **do** $unprocessed(input\_q, v) := \{\overline{x}'\}$;
while *there exists* $(u, v) \in E$ *such that* `active-edge`(u, v) *holds* **do**
    select $(u, v) \in E$ such that `active-edge`(u, v) holds;
    `// any strategy is acceptable for the above selection`
    `fire`(u, v)
return $tuples(ans\_q)$

- store the subquery $(\bar{t}, \delta)$ in $subqueries(v)$, and later, for each new $\bar{t}'$ added to $tuples(ans\_p)$, if $atom(v)\delta = B_{i,j}\delta$ is unifiable with a fresh variant of $p(\bar{t}')$ by an mgu γ then transfer the subquery $(\bar{t}\gamma, (\delta\gamma)_{|post\_vars(v)})$ through $(v, succ(v))$,

– if $v = post\_filter_i$ and p is the predicate of A_i then transfer the tuple $\bar{t}$ through $(post\_filter_i, ans\_p)$ to add it to $tuples(ans\_p)$.

Formally, the processing of a subquery is designed more sophisticatedly so that:

– every subquery or input/answer tuple that is subsumed by another one or has a term-depth greater than a fixed bound l is ignored,

– the processing is divided into smaller steps which can be delayed at each node to maximize adjustability and allow various control strategies,

– the processing is done set-at-a-time (e.g., for all the unprocessed subqueries accumulated in a given node).

The procedure `transfer`(D, u, v) (on page 26) specifies the effects of transferring data D through an edge (u, v) of a QSQ-net. If v is of the form pre_filter_i or $post\_filter_i$ or ($v = filter_{i,j}$ and $kind(v) = extensional$ and $T(v) = false$) then the input D for v is processed immediately and an appropriate data Γ is produced and transferred through $(v, succ(v))$. Otherwise, the input D for v is not processed immediately, but accumulated into the structure of v in an appropriate way.

The function `active-edge`(u, v) (on page 28) returns *true* for an edge (u, v) if data accumulated in u can be processed to produce some data to transfer through (u, v), and returns $false$ otherwise. If `active-edge`(u, v) is *true* then the procedure `fire`(u, v) (on page 28) processes the data accumulated in u that has not been processed before to transfer appropriate data through the edge (u, v). This procedure uses the procedure `transfer`(D, u, v). Both procedures `fire`(u, v) and `transfer`(D, u, v) use a parameter l as a term-depth bound for tuples and substitutions.

Algorithm 1 (on page 20) presents our QSQN evaluation method for Horn knowledge bases. It repeatedly selects an active edge and fires the operation for the edge. Such a selection is decided by the adopted control strategy, which can be arbitrary.

3.1.1 An Illustrative Example

Example 3.3. This example illustrates Algorithm 1 step by step. Consider the following Horn knowledge base (P, I) and the query $s(x)$, where p and s are intensional predicates, q is an extensional predicate, x, y, z are variables, and $a - o$, u are constant symbols:

– the positive logic program P:

$$\begin{aligned} &p(x, y) \leftarrow q(x, y) \\ &p(x, y) \leftarrow q(x, z), p(z, y) \\ &s(x) \leftarrow p(b, x) \end{aligned}$$

– the extensional instance I (illustrated in Figure 3.4):

$$\begin{aligned} I(q) = \; &\{(a, b), (b, c), (c, d), (d, e), (b, f), (f, g), (b, h), \\ &(h, g), (i, j), (j, k), (k, l), (m, n), (n, u), (n, o)\} \end{aligned}$$

– the query: $s(x)$.

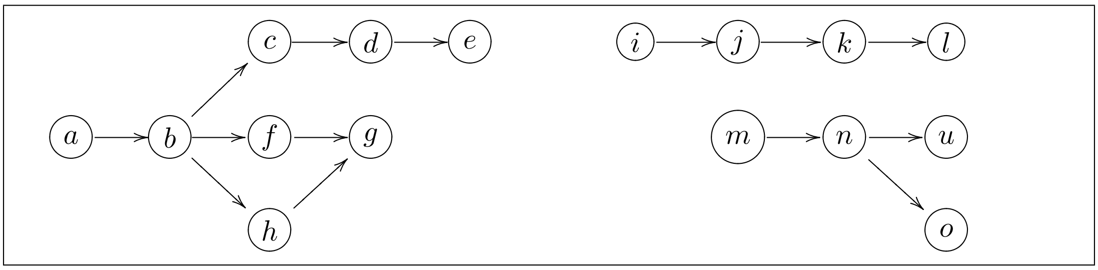


Fig. 3.4: A graph used for Example 3.3.

The QSQ topological structure of P is presented in Figure 3.5. We give below a trace of a run of Algorithm 1 that evaluates the query $(P, s(x))$ on the extensional instance I, using term-depth bound $l = 0$ and the memorizing type T that maps each node v such that $kind(v) = extensional$ (i.e., $filter_{1,1}$ and $filter_{2,1}$) to $false$. For convenience, we denote the edges of the net with names $E_1 - E_{17}$ as shown in Figure 3.5.

Algorithm 1 starts with an empty QSQ-net. It then adds a fresh variant (x_1) of (x) to the empty sets $tuples(input\_s)$ and $unprocessed(E_{14})$. Next, it repeatedly selects an active edge and fires the edge. Assume that the selection is done as follows.

1. $\mathbf{E_{14} - E_{15}}$

   After processing $unprocessed(E_{14})$, the algorithm empties this set and transfers $\{(x_1)\}$ through the edge E_{14}. This produces $\{((x_1), \{x/x_1\})\}$, which is then transferred through the edge E_{15} and added to the empty sets $subqueries(filter_{3,1})$, $unprocessed\_subqueries(filter_{3,1})$ and $unprocessed\_subqueries_2(filter_{3,1})$.

2. $\mathbf{E_{13}}$

   After processing $unprocessed\_subqueries_2(filter_{3,1})$, the algorithm empties this set and transfers $\{(b, x_1)\}$ through E_{13}. This adds a fresh variant (b, x_2) of the tuple (b, x_1) to the empty sets $tuples(input\_p)$, $unprocessed(E_1)$ and $unprocessed(E_7)$.

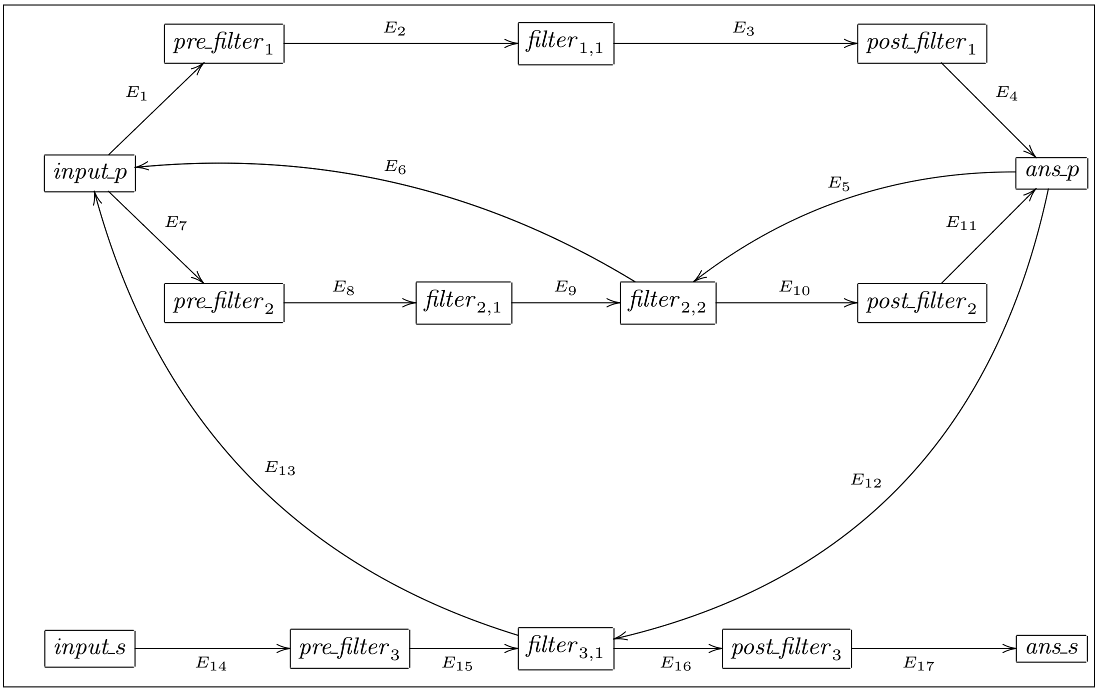


Fig. 3.5: The QSQ topological structure of the program given in Example 3.3.

input_s	*ans_s*	*input_p*	*ans_p*
x_1 **(0)**	c **(15)**	(b, x_2) **(2)**	(b, c) **(9)**
	f	(c, x_3) **(4)**	(b, f)
	h	(f, x_4)	(b, h)
	d	(h, x_5)	(c, d)
	g	(d, x_6) **(6)**	(f, g)
	e	(g, x_7)	(h, g)
		(e, x_8) **(8)**	(d, e)
			(b, d) **(11)**
			(b, g)
			(c, e)
			(b, e) **(13)**

Table 3.1: A summary of the steps at which the data (i.e., tuples) were added to *input_s*, *ans_s*, *input_p*, *ans_p*, respectively.

3. $\mathbf{E_7 - E_8 - E_9}$

   After processing *unprocessed*(E_7), the algorithm empties this set and transfers $\{(b, x_2)\}$ through the edge E_7. This produces $\{((b, x_2), \{x/b, y/x_2\})\}$, which is then transferred through the edge E_8, producing $\{((b, x_2), \{y/x_2, z/c\}), ((b, x_2), \{y/x_2, z/f\}), ((b, x_2), \{y/x_2, z/h\})\}$, which in turn is then transferred through the edge E_9 and added to the empty sets *subqueries*$(\mathit{filter}_{2,2})$, *unprocessed_subqueries*$(\mathit{filter}_{2,2})$ and *unprocessed_subqueries$_2$*$(\mathit{filter}_{2,2})$.

4. $\mathbf{E_6}$

   After processing *unprocessed_subqueries$_2$*$(\mathit{filter}_{2,2})$, the algorithm empties this set and transfers $\{(c, x_2), (f, x_2), (h, x_2)\}$ through the edge E_6. This adds fresh variants of these tuples, namely (c, x_3), (f, x_4) and (h, x_5), to the sets *tuples*$(\mathit{input\_p})$, *unprocessed*(E_1) and *unprocessed*(E_7). After these steps, we have:

   - $\mathit{unprocessed}(E_1) = \mathit{tuples}(\mathit{input\_p}) = \{(b, x_2), (c, x_3), (f, x_4), (h, x_5)\}$,
   - $\mathit{unprocessed}(E_7) = \{(c, x_3), (f, x_4), (h, x_5)\}$.

5. $\mathbf{E_7 - E_8 - E_9}$

   After processing *unprocessed*(E_7), the algorithm empties this set and transfers $\{(c, x_3), (f, x_4), (h, x_5)\}$ through the edge E_7. This produces $\{((c, x_3), \{x/c, y/x_3\}), ((f, x_4), \{x/f, y/x_4\}), ((h, x_5), \{x/h, y/x_5\})\}$, which is then transferred through the edge E_8, producing $\{((c, x_3), \{y/x_3, z/d\}), ((f, x_4), \{y/x_4, z/g\}), ((h, x_5), \{y/x_5, z/g\})\}$, which in turn is then transferred through the edge E_9 and added to the sets *subqueries*$(\mathit{filter}_{2,2})$, *unprocessed_subqueries*$(\mathit{filter}_{2,2})$ and *unprocessed_subqueries$_2$*$(\mathit{filter}_{2,2})$. After these steps, we have:

   - $\mathit{unprocessed\_subqueries}(\mathit{filter}_{2,2}) = \mathit{subqueries}(\mathit{filter}_{2,2}) =$
     $\{((b, x_2), \{y/x_2, z/c\}), ((b, x_2), \{y/x_2, z/f\}), ((b, x_2), \{y/x_2, z/h\}),$
     $((c, x_3), \{y/x_3, z/d\}), ((f, x_4), \{y/x_4, z/g\}), ((h, x_5), \{y/x_5, z/g\})\}$,
   - $\mathit{unprocessed\_subqueries}_2(\mathit{filter}_{2,2}) =$
     $\{((c, x_3), \{y/x_3, z/d\}), ((f, x_4), \{y/x_4, z/g\}), ((h, x_5), \{y/x_5, z/g\})\}$.

6. $\mathbf{E_6}$

   After processing *unprocessed_subqueries$_2$*$(\mathit{filter}_{2,2})$, the algorithm empties this set and transfers $\{(d, x_3), (g, x_4)\}$ through the edge E_6. This adds fresh variants of these tuples, namely (d, x_6) and (g, x_7), to the sets *tuples*$(\mathit{input\_p})$, *unprocessed*(E_1) and *unprocessed*(E_7). After these steps, we have:

   - $\mathit{unprocessed}(E_1) = \mathit{tuples}(\mathit{input\_p}) = \{(b, x_2), (c, x_3), (f, x_4), (h, x_5), (d, x_6), (g, x_7)\}$,
   - $\mathit{unprocessed}(E_7) = \{(d, x_6), (g, x_7)\}$.

7. $\mathbf{E_7 - E_8 - E_9}$

   After processing *unprocessed*(E_7), the algorithm empties this set and transfers $\{(d, x_6), (g, x_7)\}$ through the edge E_7. This produces $\{((d, x_6), \{x/d, y/x_6\}), ((g, x_7), \{x/g, y/x_7\})\}$, which is then transfers through the edge E_8, producing $\{((d, x_6), \{y/x_6, z/e\})\}$, which in turn is then transferred through the edge E_9 and added to the sets *subqueries*$(\mathit{filter}_{2,2})$, *unprocessed_subqueries*$(\mathit{filter}_{2,2})$ and *unprocessed_subqueries$_2$*$(\mathit{filter}_{2,2})$. After these steps, we have:

– $unprocessed\_subqueries(filter_{2,2}) = subqueries(filter_{2,2}) = \{((b, x_2), \{y/x_2, z/c\}),$
$((b, x_2), \{y/x_2, z/f\}), ((b, x_2), \{y/x_2, z/h\}), ((c, x_3), \{y/x_3, z/d\}),$
$((f, x_4), \{y/x_4, z/g\}), ((h, x_5), \{y/x_5, z/g\}), ((d, x_6), \{y/x_6, z/e\})\},$
– $unprocessed\_subqueries_2(filter_{2,2}) = \{((d, x_6), \{y/x_6, z/e\})\}.$

8. $\mathbf{E_6}$

   After processing $unprocessed\_subqueries_2(filter_{2,2})$, the algorithm empties this set and transfers $\{(e, x_6)\}$ through the edge E_6. This adds a fresh variant (e, x_8) of the tuple $\{(e, x_6)\}$ to the sets $tuples(input\_p)$, $unprocessed(E_1)$ and $unprocessed(E_7)$. After these steps, we have:

   – $unprocessed(E_1) = tuples(input\_p) =$
   $\{(b, x_2), (c, x_3), (f, x_4), (h, x_5), (d, x_6), (g, x_7), (e, x_8)\},$
   – $unprocessed(E_7) = \{(e, x_8)\}.$

9. $\mathbf{E_1 - E_2 - E_3 - E_4}$

   After processing $unprocessed(E_1)$, the algorithm empties this set and transfers $\{(b, x_2), (c, x_3), (f, x_4), (h, x_5), (d, x_6), (g, x_7), (e, x_8)\}$ through the edge E_1. This produces $\{((b, x_2), \{x/b, y/x_2\}), ((c, x_3), \{x/c, y/x_3\}), ((f, x_4), \{x/f, y/x_4\}), ((h, x_5), \{x/h, y/x_5\}), ((d, x_6), \{x/d, y/x_6\}), ((g, x_7), \{x/g, y/x_7\}), ((e, x_8), \{x/e, y/x_8\})\}$, which is then transferred through the edge E_2, producing $\{((b, c), \varepsilon), ((b, f), \varepsilon), ((b, h), \varepsilon), ((c, d), \varepsilon), ((f, g), \varepsilon), ((h, g), \varepsilon), ((d, e), \varepsilon)\}$, which in turn is then transferred through the edge E_3, producing $\{(b, c), (b, f), (b, h), (c, d), (f, g), (h, g), (d, e)\}$, which in turn is then transferred through the edge E_4 and added to the empty sets $tuples(ans\_p)$, $unprocessed(E_5)$ and $unprocessed(E_{12})$.

10. $\mathbf{E_5}$

    After processing $unprocessed(E_5)$, the algorithm empties this set and transfers $\{(b, c), (b, f), (b, h), (c, d), (f, g), (h, g), (d, e)\}$ through the edge E_5 and adds these tuples to the empty set $unprocessed\_tuples(filter_{2,2})$.

11. $\mathbf{E_{10} - E_{11}}$

    After processing $unprocessed\_tuples(filter_{2,2})$ and $unprocessed\_subqueries(filter_{2,2})$, the algorithm empties these sets and transfers $\{((b, d), \varepsilon), ((b, g), \varepsilon), ((c, e), \varepsilon)\}$ through the edge E_{10}. This produces $\{(b, d), (b, g), (c, e)\}$, which is then transferred through the edge E_{11} and added to the sets $tuples(ans\_p)$, $unprocessed(E_5)$ and $unprocessed(E_{12})$. After these steps, we have:

    – $unprocessed(E_{12}) = tuples(ans\_p) =$
    $\{(b, c), (b, f), (b, h), (c, d), (f, g), (h, g), (d, e), (b, d), (b, g), (c, e)\},$
    – $unprocessed(E_5) = \{(b, d), (b, g), (c, e)\}.$

12. $\mathbf{E_5}$

    After processing $unprocessed(E_5)$, the algorithm empties this set and transfers $\{(b, d), (b, g), (c, e)\}$ through the edge E_5 and adds these tuples to the empty set $unprocessed\_tuples(filter_{2,2})$.

13. $\mathbf{E_{10} - E_{11}}$

    After processing $unprocessed\_tuples(filter_{2,2})$, the algorithm empties this set and transfers $\{((b, e), \varepsilon)\}$ through the edge E_{10}. This produces $\{(b, e)\}$, which is then transferred through the edge E_{11} and added to the sets *tuples*(ans_p), *unprocessed*(E_5) and *unprocessed*(E_{12}). After these steps, we have:

- *unprocessed*(E_{12}) = *tuples*(ans_p) =
  $\{(b,c),\ (b,f),\ (b,h),\ (c,d),\ (f,g),\ (h,g),\ (d,e),\ (b,d),\ (b,g),\ (c,e),\ (b,e)\}$,
- *unprocessed*(E_5) = $\{(b,e)\}$.

14. $\mathbf{E_{12}}$

    After processing *unprocessed*(E_{12}), the algorithm empties this set and transfers $\{(b,c),\ (b,f),\ (b,h),\ (c,d),\ (f,g),\ (h,g),\ (d,e),\ (b,d),\ (b,g),\ (c,e),\ (b,e)\}$ through the edge E_{12} and adds these tuples to the empty set *unprocessed_tuples*($filter_{3,1}$).

15. $\mathbf{E_{16} - E_{17}}$

    After processing *unprocessed_tuples*($filter_{3,1}$) and *unprocessed_subqueries*($filter_{3,1}$), the algorithm empties these sets and transfers $\{((c),\varepsilon),\ ((f),\varepsilon),\ ((h),\varepsilon),\ ((d),\varepsilon),\ ((g),\varepsilon),\ ((e),\varepsilon)\}$ through the edge E_{16}. This produces $\{(c),\ (f),\ (h),\ (d),\ (g),\ (e)\}$, which is then transferred through the edge E_{17} and added to the empty set *tuples*(ans_s).

16. $\mathbf{E_5},\ \mathbf{E_7},\ \mathbf{E_{10}}$

    The edges E_5 and E_7 are still active, with *unprocessed*(E_5) $= \{(b,e)\}$ and *unprocessed*(E_7) $= \{(e,x_8)\}$. Firing the edge E_5 causes the edge E_{10} to become active, but after that, firing the edges E_7 and E_{10} does not create data to be transferred.

At this point, no edges are active (in particular, all the attributes *unprocessed*, *unprocessed_subqueries*, *unprocessed_subqueries*$_2$ and *unprocessed_tuples* of the nodes in the net are empty sets). The algorithm terminates and returns the set *tuples*(ans_s) $= \{(c),(f),(h),(d),(g),(e)\}$.

Table 3.1 summarizes the effects of the steps of this trace. The numbers in bold font indicate the corresponding steps of the trace, which are listed in Example 3.3. ∎

3.1.2 Relaxing Term-Depth Bound

Suppose that we want to compute as many as possible but no more than k correct answers for a query $(P, q(\overline{x}))$ on an extensional instance I within time limit L. Then we can use iterative deepening search which iteratively increases term-depth bound for atoms and substitutions occurring in the computation as follows:

1. Initialize term-depth bound l to 0 (or another small natural number).
2. Run Algorithm 1 for evaluating $(P, q(\overline{x}))$ on I within the time limit.
3. While *tuples*(ans_q) contains less than k tuples and the time limit was not reached yet, do:
   (a) Clear (empty) all the sets of the form *tuples*($input\_p$) and *subqueries*($filter_{i,j}$).
   (b) Increase term-depth bound l by 1.
   (c) Run Algorithm 1 without Steps 1 and 2.
4. Return *tuples*(ans_q).

Procedure transfer(D, u, v)

Global data: a Horn knowledge base (P, I), a QSQ-net $N = (V, E, T, C)$ of P, and a term-depth bound l.

Input: data D to transfer through the edge $(u, v) \in E$.

if $D = \emptyset$ **then return**;
if *u is input_p* **then**
  $\Gamma := \emptyset$;
  foreach $\bar{t} \in D$ **do**
    if *$p(\bar{t})$ and $atom(v)$ are unifiable by an mgu γ* **then**
      `add-subquery`$(\bar{t}\gamma, \gamma_{|post\_vars(v)}, \Gamma, succ(v))$
  `transfer`$(\Gamma, v, succ(v))$
else if *u is ans_p* **then** $unprocessed\_tuples(v) := unprocessed\_tuples(v) \cup D$;
else if *v is input_p or ans_p* **then**
  foreach $\bar{t} \in D$ **do**
    let $\bar{t}'$ be a fresh variant of $\bar{t}$;
    if *$\bar{t}'$ is not an instance of any tuple from $tuples(v)$* **then**
      foreach $\bar{t}'' \in tuples(v)$ **do**
        if *$\bar{t}''$ is an instance of $\bar{t}'$* **then**
          delete $\bar{t}''$ from $tuples(v)$;
          foreach $(v, w) \in E$ **do** delete $\bar{t}''$ from $unprocessed(v, w)$;
      if *v is input_p* **then**
        add $\bar{t}'$ to $tuples(v)$;
        foreach $(v, w) \in E$ **do** add $\bar{t}'$ to $unprocessed(v, w)$;
      else
        add $\bar{t}$ to $tuples(v)$;
        foreach $(v, w) \in E$ **do** add $\bar{t}$ to $unprocessed(v, w)$;
else if *v is $filter_{i,j}$ and $kind(v) = extensional$ and $T(v) = false$* **then**
  let $p = pred(v)$ and set $\Gamma := \emptyset$;
  foreach $(\bar{t}, \delta) \in D$ **do**
    if `term-depth`$(atom(v)\delta) \leq l$ **then**
      foreach $\bar{t}' \in I(p)$ **do**
        if *$atom(v)\delta$ is unifiable with a fresh variant of $p(\bar{t}')$ by an mgu γ* **then**
          `add-subquery`$(\bar{t}\gamma, (\delta\gamma)_{|post\_vars(v)}, \Gamma, succ(v))$
  `transfer`$(\Gamma, v, succ(v))$
else if *v is $filter_{i,j}$ and ($kind(v) = extensional$ and $T(v) = true$ or $kind(v) = intensional$)* **then**
  foreach $(\bar{t}, \delta) \in D$ **do**
    if `term-depth`$(atom(v)\delta) \leq l$ **then**
      if *no subquery in $subqueries(v)$ is more general than $(\bar{t}, \delta)$* **then**
        delete from $subqueries(v)$ all subqueries less general than $(\bar{t}, \delta)$;
        delete from $unprocessed\_subqueries(v)$ all subqueries less general than $(\bar{t}, \delta)$;
        add $(\bar{t}, \delta)$ to both $subqueries(v)$ and $unprocessed\_subqueries(v)$;
        if $kind(v) = intensional$ **then**
          delete from $unprocessed\_subqueries_2(v)$ all subqueries less general than $(\bar{t}, \delta)$;
          add $(\bar{t}, \delta)$ to $unprocessed\_subqueries_2(v)$
else // v is of the form $post\_filter_i$
  $\Gamma := \{\bar{t} \mid (\bar{t}, \varepsilon) \in D\}$;
  `transfer`$(\Gamma, v, succ(v))$

Procedure add-subquery($\bar{t}, \delta, \Gamma, v$)

Purpose: add the subquery $(\bar{t}, \delta)$ to Γ, but keep in Γ only the most general subqueries w.r.t. v.

```
if term-depth(t̄) ≤ l and term-depth(δ) ≤ l and no subquery in Γ is more general than (t̄, δ) w.r.t. v then
    delete from Γ all subqueries less general than (t̄, δ) w.r.t. v;
    add (t̄, δ) to Γ
```

Procedure add-tuple($\bar{t}, \Gamma$)

Purpose: add the tuple $\bar{t}$ to Γ, but keep in Γ only the most general tuples.

```
let t̄' be a fresh variant of t̄;
if t̄' is not an instance of any tuple from Γ then
    delete from Γ all tuples that are instances of t̄';
    add t̄' to Γ
```

3.2 Properties of Algorithm 1

We present below properties of Algorithm 1, which were first proved by Nguyen in [45][1]. As QSQN is a special case of QSQN-TRE specified in the next chapter[2], they follow from the corresponding properties of QSQN-TRE, which are specified and proved in Chapter 4.

Soundness: After a run of Algorithm 1 on a query $(P, q(\overline{x}))$ and an extensional instance I, for every intensional predicate p of P, every computed answer $\bar{t} \in \mathit{tuples}(\mathit{ans\_p})$ is a correct answer in the sense that $P \cup I \models \forall(p(\bar{t}))$. ∎

Completeness: After a run of Algorithm 1 (using parameter l) on a query $(P, q(\overline{x}))$ and an extensional instance I, for every SLD-refutation of $P \cup I \cup \{\leftarrow q(\overline{x})\}$ that uses the leftmost selection function, does not contain any goal with term-depth greater than l and has a computed answer θ with term-depth not greater than l, there exists $\overline{s} \in \mathit{tuples}(\mathit{ans\_q})$ such that $\overline{x}\theta$ is an instance of a variant of $\overline{s}$. ∎

Together with Theorem 2.1 (on the completeness of SLD-resolution), this property makes a relationship between correct answers for $P \cup I \cup \{\leftarrow q(\overline{x})\}$ and the answers computed by Algorithm 1 for the query $(P, q(\overline{x}))$ on the extensional instance I.

For queries and extensional instances without function symbols, we take term-depth bound $l = 0$ and obtain the following completeness result, which immediately follows from the above property.

[1]The proofs given in [45] were later improved by me and the corresponding revision is available at [12].

[2]QSQN-TRE is the same as QSQN when $T(p) = \mathit{false}$ for every intensional predicate p used in P, where T is a function used in the definition of the QSQN-TRE structure.

Function active-edge(u, v)

Global data: a QSQ-net $N = (V, E, T, C)$.
Input: an edge $(u, v) \in E$.
Output: *true* if there is data to transfer through the edge (u, v), and $false$ otherwise.

if u *is* $\mathit{pre\_filter}_i$ *or* $\mathit{post\_filter}_i$ **then return** $false$;
else if u *is* $\mathit{input\_p}$ *or* $\mathit{ans\_p}$ **then return** $\mathit{unprocessed}(u, v) \neq \emptyset$;
else if u *is* $\mathit{filter}_{i,j}$ *and* $\mathit{kind}(u) = \mathit{extensional}$ **then**
  return $T(u) = \mathit{true} \wedge \mathit{unprocessed\_subqueries}(u) \neq \emptyset$
else `//` u `is of the form` $\mathit{filter}_{i,j}$ `and` $\mathit{kind}(u) = \mathit{intensional}$
  let $p = \mathit{pred}(u)$;
  if $v = \mathit{input\_p}$ **then return** $\mathit{unprocessed\_subqueries}_2(u) \neq \emptyset$;
  else return $\mathit{unprocessed\_subqueries}(u) \neq \emptyset \vee \mathit{unprocessed\_tuples}(u) \neq \emptyset$;

Procedure fire(u, v)

Global data: a Horn knowledge base (P, I), a QSQ-net $N = (V, E, T, C)$ of P, and a term-depth bound l.
Input: an edge $(u, v) \in E$ such that `active-edge`(u, v) holds.

if u *is* $\mathit{input\_p}$ *or* $\mathit{ans\_p}$ **then**
  `transfer`$(\mathit{unprocessed}(u, v), u, v)$;
  $\mathit{unprocessed}(u, v) := \emptyset$
else if u *is* $\mathit{filter}_{i,j}$ *and* $\mathit{kind}(u) = \mathit{extensional}$ *and* $T(u) = \mathit{true}$ **then**
  let $p = \mathit{pred}(u)$ and set $\Gamma := \emptyset$;
  foreach $(\bar{t}, \delta) \in \mathit{unprocessed\_subqueries}(u)$ **do**
    foreach $\bar{t}' \in I(p)$ **do**
      if $\mathit{atom}(u)\delta$ *is unifiable with a fresh variant of* $p(\bar{t}')$ *by an mgu* γ **then**
        `add-subquery`$(\bar{t}\gamma, (\delta\gamma)_{|\mathit{post\_vars}(u)}, \Gamma, v)$
  $\mathit{unprocessed\_subqueries}(u) := \emptyset$;
  `transfer`(Γ, u, v)
else if u *is* $\mathit{filter}_{i,j}$ *and* $\mathit{kind}(u) = \mathit{intensional}$ **then**
  let $p = \mathit{pred}(u)$ and set $\Gamma := \emptyset$;
  if $v = \mathit{input\_p}$ **then**
    foreach $(\bar{t}, \delta) \in \mathit{unprocessed\_subqueries}_2(u)$ **do** let $p(\bar{t}') = \mathit{atom}(u)\delta$, `add-tuple`$(\bar{t}', \Gamma)$;
    $\mathit{unprocessed\_subqueries}_2(u) := \emptyset$;
  else
    foreach $(\bar{t}, \delta) \in \mathit{unprocessed\_subqueries}(u)$ **do**
      foreach $\bar{t}' \in \mathit{tuples}(\mathit{ans\_p})$ **do**
        if $\mathit{atom}(u)\delta$ *is unifiable with a fresh variant of* $p(\bar{t}')$ *by an mgu* γ **then**
          `add-subquery`$(\bar{t}\gamma, (\delta\gamma)_{|\mathit{post\_vars}(u)}, \Gamma, v)$
    $\mathit{unprocessed\_subqueries}(u) := \emptyset$;
    foreach $\bar{t} \in \mathit{unprocessed\_tuples}(u)$ **do**
      foreach $(\bar{t}', \delta) \in \mathit{subqueries}(u)$ **do**
        if $\mathit{atom}(u)\delta$ *is unifiable with a fresh variant of* $p(\bar{t})$ *by an mgu* γ **then**
          `add-subquery`$(\bar{t}'\gamma, (\delta\gamma)_{|\mathit{post\_vars}(u)}, \Gamma, v)$
    $\mathit{unprocessed\_tuples}(u) := \emptyset$
  `transfer`(Γ, u, v)

After a run of Algorithm 1 using $l = 0$ on a query $(P, q(\overline{x}))$ and an extensional instance I that do not contain function symbols, for every computed answer θ of an SLD-refutation of $P \cup I \cup \{\leftarrow q(\overline{x})\}$ that uses the leftmost selection function, there exists $\overline{t} \in tuples(ans\_q)$ such that $\overline{x}\theta$ is an instance of a variant of $\overline{t}$.

Data Complexity: For a fixed query and a fixed bound l on term-depth, Algorithm 1 runs in polynomial time in the size of the extensional instance. ∎

Chapter 4

Incorporating Tail-Recursion Elimination into QSQN

Query optimization has received much attention from researchers in the database community. Several optimization methods and techniques have been developed to improve performance of query evaluation. One of them is to reduce the number of materialized intermediate results during the processing by using the tail-recursion elimination. The general form of recursion requires the compiler to allocate storage on the stack at runtime. Such a memory consumption may be costly. A call is tail-recursive if no work remains to be done after the call returns. Tail recursion is a special case of recursion that is semantically equivalent to the iteration construct. A tail-recursive program can be compiled as efficiently as iterative programs by applying tail-recursion elimination. Ross' work [53] contains a very good example about the usefulness of tail-recursion elimination. Let's consider a slightly modified version of that example.

Example 4.1. Let P be the positive logic program consisting of the following clauses:

$$\begin{array}{l} p(x,y) \leftarrow e(x,z), p(z,y) \\ p(m,x) \leftarrow t(x) \end{array}$$

where p is an intensional predicate, e and t are extensional predicates, m is a natural number (a constant) and x, y, z are variables. Let $p(1,x)$ be the query, n a natural number, and let the extensional instance I for e and t be as follows:

$$\begin{array}{rcl} I(e) & = & \{(1,2),(2,3),\ldots,(m-1,m),(m,1)\}, \\ I(t) & = & \{1,\ldots,n\}. \end{array}$$

To make this example more concrete, suppose that: $e(x,z)$ holds when there is a way to get from town x to town z, where the towns are numbered from 1 to m and m denotes the capital; $t(x)$ holds when item x is available in the capital; items are numbered from 1 to n and all items are available in the capital; $p(z,y)$ holds if it is possible to get from town z to a town that has item y. For the query $p(1,x)$, the task is to find all available items starting from town 1.

To answer the query, methods such as QSQR, QSQN, Magic-Sets would evaluate every subquery of the form $p(i,x)$, where $1 \leq i \leq m$, and thus store $m \times n$ tuples (i,j)

in the answer relation for p, where $1 \leq i \leq m$ and $1 \leq j \leq n$. As can be seen, for answering the query $p(1, x)$, we do not need to store the intermediate answer tuples (i, j) with $i > 1$ for p if we apply tail-recursion elimination. We only need to store n answer tuples $(1, j)$ with $1 \leq j \leq n$ and m subqueries (i, x) with $1 \leq i \leq m$ for p. That is, we need to store only $m + n$ instead of $m \times n$ tuples. The example in Ross' work [53] considers $m = 100$ towns and $n = 1000$ items, and it is easy to see how big the difference is. ∎

In Chapter 3, we formulated the query-subquery nets and used them to develop the first framework for developing algorithms for evaluating queries to Horn knowledge bases. The framework forms a generic evaluation method called QSQN. The experimental results in Section 6.2 for QSQN indicate the usefulness of this method. It is desirable to study how to develop the other evaluation methods that are based on query-subquery nets.

In this chapter, we first incorporate tail-recursion elimination into query-subquery nets in order to formulate the QSQN-TRE evaluation method for Horn knowledge bases. We then present another method called QSQN-rTRE, which can eliminate not only tail-recursive predicates but also intensional predicates that appear rightmost in the bodies of the program clauses.

The rest of this chapter is structured as follows. Section 4.1 presents the QSQN-TRE evaluation method for Horn knowledge bases together with its properties and an illustrative example. Section 4.2 discusses the QSQN-rTRE evaluation method and its properties. The preliminary experiments and a discussion for the QSQN-TRE and QSQN-rTRE methods are presented later in Sections 6.3 and 6.4, respectively.

4.1 QSQN with Tail-Recursion Elimination

This section presents a method called QSQN-TRE for evaluating queries to Horn knowledge bases by integrating query-subquery nets with a form of tail-recursion elimination. The aim is to reduce materializing the intermediate results during the processing of a query with tail-recursion.

4.1.1 Definitions

Let P be a positive logic program and $\varphi_1, \ldots, \varphi_m$ be all the program clauses of P, with $\varphi_i = (A_i \leftarrow B_{i,1}, \ldots, B_{i,n_i})$, for $1 \leq i \leq m$ and $n_i \geq 0$.

Definition 4.1 (Tail-Recursion). A program clause $\varphi_i = (A_i \leftarrow B_{i,1}, \ldots, B_{i,n_i})$, for $n_i > 0$, is said to be *recursive* whenever some $B_{i,j}$ $(1 \leq j \leq n_i)$ has the same predicate as A_i. If B_{i,n_i} has the same predicate as A_i then the clause is *tail-recursive* and in this case the predicate of B_{i,n_i} is a *tail-recursive predicate*. ∎

The following definition shows how to make a QSQN-TRE structure from the given program P.

Definition 4.2 (QSQN-TRE Structure). A *query-subquery net structure with tail-recursion elimination* (*QSQN-TRE structure* for short) of P is a tuple (V, E, T) such that:

- T is a pair (T_{edb}, T_{idb}), called the *type* of the net structure.
- T_{idb} is a function that maps each intensional predicate to *true* or *false*. (If $T_{idb}(p) = true$ then the intensional relation p will be computed using tail-recursion elimination[1]. T_{edb} will be explained shortly.)
- V is a set of nodes that includes:
  - $input\_p$ and ans_p, for each intensional predicate p of P,
  - pre_filter_i, $filter_{i,1}$, ..., $filter_{i,n_i}$, for each $1 \leq i \leq m$,
  - $post\_filter_i$ if either φ_i is not tail-recursive or $T_{idb}(p) = false$, for each $1 \leq i \leq m$, where p is the predicate of A_i.
- E is a set of edges that includes:
  - $(input\_p, pre\_filter_i)$, for each $1 \leq i \leq m$, where p is the predicate of A_i,
  - $(pre\_filter_i, filter_{i,1})$, for each $1 \leq i \leq m$ such that $n_i \geq 1$,
  - $(filter_{i,1}, filter_{i,2})$, ..., $(filter_{i,n_i-1}, filter_{i,n_i})$, for each $1 \leq i \leq m$,
  - $(filter_{i,n_i}, post\_filter_i)$, for each $1 \leq i \leq m$ such that $n_i \geq 1$ and $post\_filter_i$ exists,
  - $(pre\_filter_i, post\_filter_i)$, for each $1 \leq i \leq m$ such that $n_i = 0$,
  - $(post\_filter_i, ans\_p)$, for each $1 \leq i \leq m$ such that $post\_filter_i$ exists, where p is the predicate of A_i,
  - $(filter_{i,j}, input\_p)$, for each $1 \leq i \leq m$ and $1 \leq j \leq n_i$ such that the predicate p of $B_{i,j}$ is an intensional predicate,
  - $(ans\_p, filter_{i,j})$, for each intensional predicate p and $1 \leq i \leq m$ and $1 \leq j \leq n_i$ such that $B_{i,j}$ is an atom of p and either $(1 \leq j < n_i)$ or $(j = n_i$ and $post\_filter_i$ exists).
- T_{edb} is a function that maps each $filter_{i,j} \in V$ such that the predicate of $B_{i,j}$ is extensional to *true* or *false*. (If $T_{edb}(filter_{i,j}) = false$ then subqueries for $filter_{i,j}$ are always processed immediately without being accumulated at $filter_{i,j}$). ∎

From now on, $T(v)$ denotes $T_{edb}(v)$ if v is a node $filter_{i,j}$ such that $B_{i,j}$ is an extensional predicate, and $T(p)$ denotes $T_{idb}(p)$ for an intensional predicate p. Thus, T can be called a *memorizing type* for extensional predicates (as in QSQ-net structures), and a *tail-recursion-elimination type* for intensional predicates.

We call the pair (V, E) the *QSQN-TRE topological structure* of P w.r.t. T_{idb}. The lower part of Figure 4.1 illustrates the QSQN-TRE topological structure of the positive logic program given in Example 3.1 w.r.t. the T_{idb} with $T_{idb}(p) = true$ in comparison with the QSQ topological structure of the same logic program, which is described in the upper part of this figure.

Definition 4.3 (QSQN-TRE). A *query-subquery net with tail-recursion elimination* (*QSQN-TRE* for short) of P is a tuple $N = (V, E, T, C)$ such that (V, E, T) is

[1]It is desirable to expect that $T_{idb}(p) = true$ iff p is tail-recursive w.r.t. P. However, we do not require this condition. In particular, it is possible that $T_{idb}(p) = false$ for some tail-recursive predicates p.

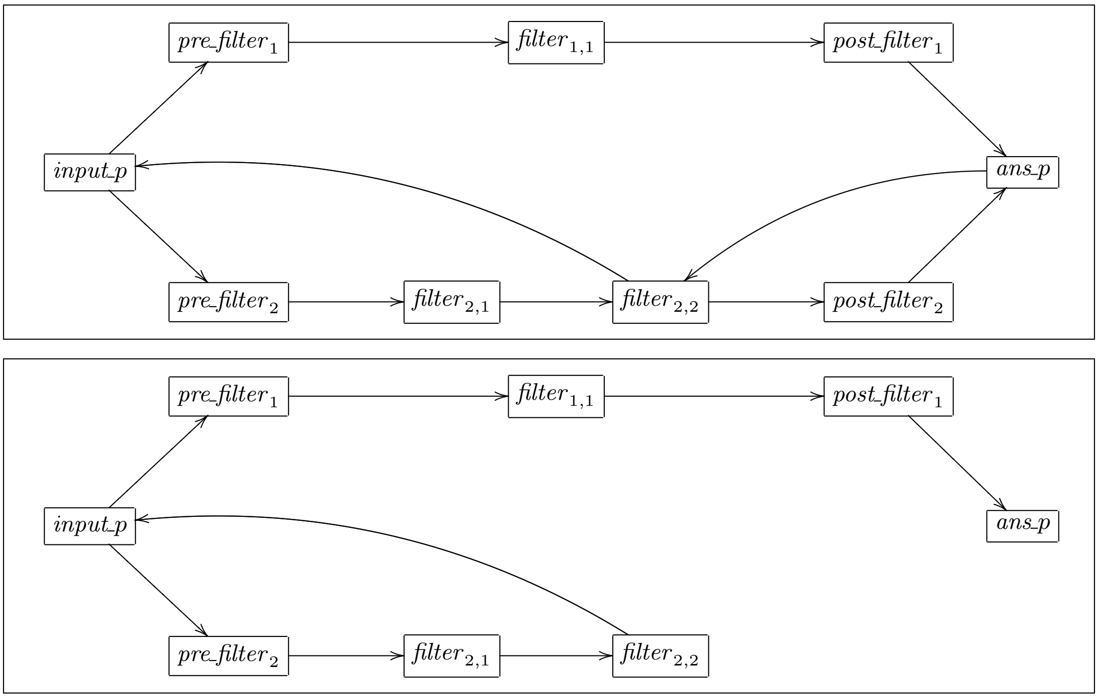


Fig. 4.1: The QSQ topological structure and the QSQN-TRE topological structure of the program given in Example 3.1.

a QSQN-TRE structure of P, C is a mapping that associates each node $v \in V$ with a structure called the *contents* of v, which differs from the one for QSQN (see Definition 3.2) in that:

- If $v = input\_p$ and $T(p) = true$ then $C(v)$ consists of:
  - $tuple\_pairs(v)$: a set of pairs of generalized tuples of the same arity as p,
  - $unprocessed(v, w)$ for each $(v, w) \in E$: a subset of $tuple\_pairs(v)$.
- If $v = filter_{i,n_i}$, $kind(v) = intensional$, $pred(v) = p$ and $T(p) = true$ then $unprocessed\_subqueries(v)$ and $unprocessed\_tuples(v)$ are empty (and can thus be ignored).

A QSQN-TRE of P is *empty* if all the sets of the form $tuple\_pairs(v)$, $tuples(v)$, $unprocessed(v, w)$, $subqueries(v)$, $unprocessed\_subqueries(v)$, $unprocessed\_subqueries_2(v)$ or $unprocessed\_tuples(v)$ are empty. ■

If $(v, w) \in E$ then w is referred to as a *successor* of v. Observe that:

- if $v \in \{pre\_filter_i, post\_filter_i\}$ or ($v = filter_{i,j}$ and $kind(v) = extensional$) then v has exactly one successor, which we denote by $succ(v)$,
- if v is $filter_{i,n_i}$ with $kind(v) = intensional$, $pred(v) = p$ and $T(p) = true$ then v has exactly one successor, which we denote by $succ_2(v) = input\_p$,
- if v is $filter_{i,j}$ with $kind(v) = intensional$, $pred(v) = p$ and either $j < n_i$ or $T(p) = false$ then v has exactly two successors: $succ(v) = filter_{i,j+1}$ if $j < n_i$; $succ(v) = post\_filter_i$ otherwise; and $succ_2(v) = input\_p$.

Figure 4.2 illustrates a QSQN-TRE (V, E, T, C) of the positive logic program given in Example 3.1 with $T(p) = \mathit{true}$.

Recall that a *subquery* is a pair of the form $(\bar{t}, \delta)$, where $\bar{t}$ is a generalized tuple and δ is an idempotent substitution such that $\mathit{dom}(\delta) \cap \mathit{Vars}(\bar{t}) = \emptyset$. The set $\mathit{unprocessed\_subqueries}_2(v)$ (resp. $\mathit{unprocessed\_subqueries}(v)$) contains the subqueries that were not transferred through the edge $(v, \mathit{succ}_2(v))$ (resp. $(v, \mathit{succ}(v))$ – when it exists).

Remark 4.1. *For an intensional predicate p with $T(p) = \mathit{true}$, the intuition behind a pair $(\bar{t}, \bar{t}') \in \mathit{tuple\_pairs}(\mathit{input\_p})$ is that:*

- *$\bar{t}$ is a usual input tuple for p, but the intended goal at a higher level is $\leftarrow p(\bar{t}')$,*
- *any correct answer for $P \cup I \cup \{\leftarrow p(\bar{t})\}$ is also a correct answer for $P \cup I \cup \{\leftarrow p(\bar{t}')\}$,*
- *if a substitution θ is a computed answer of $P \cup I \cup \{\leftarrow p(\bar{t})\}$ then we will store in $\mathit{ans\_p}$ the tuple $\bar{t}'\theta$ instead of $\bar{t}\theta$.* ■

We say that a tuple pair $(\bar{t}, \bar{t}')$ is *more general* than $(\bar{t}_2, \bar{t}_2')$, and $(\bar{t}_2, \bar{t}_2')$ is an *instance* of $(\bar{t}, \bar{t}')$, if there exists a substitution θ such that $(\bar{t}, \bar{t}')\theta = (\bar{t}_2, \bar{t}_2')$.

For $v = \mathit{filter}_{i,j}$ and p being the predicate of A_i, the meaning of a subquery $(\bar{t}, \delta) \in \mathit{subqueries}(v)$ is as follows: if $T(p) = \mathit{false}$ (resp. $T(p) = \mathit{true}$) then there exists $\bar{s} \in \mathit{tuples}(\mathit{input\_p})$ (resp. $(\bar{s}, \bar{s}') \in \mathit{tuple\_pairs}(\mathit{input\_p})$) such that for processing the goal $\leftarrow p(\bar{s})$ using the program clause $\varphi_i = (A_i \leftarrow B_{i,1}, \ldots, B_{i,n_i})$, unification of $p(\bar{s})$ and A_i as well as processing of the subgoals $B_{i,1}, \ldots, B_{i,j-1}$ were done, amongst others, by using a sequence of mgu's $\gamma_0, \ldots, \gamma_{j-1}$ with the property that $\bar{t} = \bar{s}\gamma_0 \ldots \gamma_{j-1}$ (resp. $\bar{t} = \bar{s}'\gamma_0 \ldots \gamma_{j-1}$) and $\delta = (\gamma_0 \ldots \gamma_{j-1})_{|\mathit{Vars}((B_{i,j}, \ldots, B_{i,n_i}))}$.

Informally, a subquery $(\bar{t}, \delta)$ transferred through an edge to v is processed as follows:

- If $v = \mathit{filter}_{i,j}$, $\mathit{kind}(v) = \mathit{extensional}$ and $\mathit{pred}(v) = p$ then, for each $\bar{t}' \in I(p)$, if $\mathit{atom}(v)\delta = B_{i,j}\delta$ is unifiable with a fresh variant of $p(\bar{t}')$ by an mgu γ then transfer the subquery $(\bar{t}\gamma, (\delta\gamma)_{|\mathit{post\_vars}(v)})$ through $(v, \mathit{succ}(v))$.
- If $v = \mathit{filter}_{i,j}$, $\mathit{kind}(v) = \mathit{intensional}$, $\mathit{pred}(v) = p$ and either $T(p) = \mathit{false}$ or $j < n_i$ or p is not the predicate of A_i then
    - if $T(p) = \mathit{false}$ then transfer the tuple $\bar{t}'$ such that $p(\bar{t}') = \mathit{atom}(v)\delta = B_{i,j}\delta$ through $(v, \mathit{input\_p})$ to add its fresh variant to $\mathit{tuples}(\mathit{input\_p})$,
    - else if $j < n_i$ or p is not the predicate of A_i then transfer the tuple pair $(\bar{t}', \bar{t}')$ such that $p(\bar{t}') = \mathit{atom}(v)\delta = B_{i,j}\delta$ through $(v, \mathit{input\_p})$ to add its fresh variant to $\mathit{tuple\_pairs}(\mathit{input\_p})$,
    - for each currently existing $\bar{t}' \in \mathit{tuples}(\mathit{ans\_p})$, if $\mathit{atom}(v)\delta = B_{i,j}\delta$ is unifiable with a fresh variant of $p(\bar{t}')$ by an mgu γ then transfer the subquery $(\bar{t}\gamma, (\delta\gamma)_{|\mathit{post\_vars}(v)})$ through $(v, \mathit{succ}(v))$,
    - store the subquery $(\bar{t}, \delta)$ in $\mathit{subqueries}(v)$, and later, for each new $\bar{t}'$ added to $\mathit{tuples}(\mathit{ans\_p})$, if $\mathit{atom}(v)\delta = B_{i,j}\delta$ is unifiable with a fresh variant of $p(\bar{t}')$ by an mgu γ then transfer the subquery $(\bar{t}\gamma, (\delta\gamma)_{|\mathit{post\_vars}(v)})$ through $(v, \mathit{succ}(v))$.

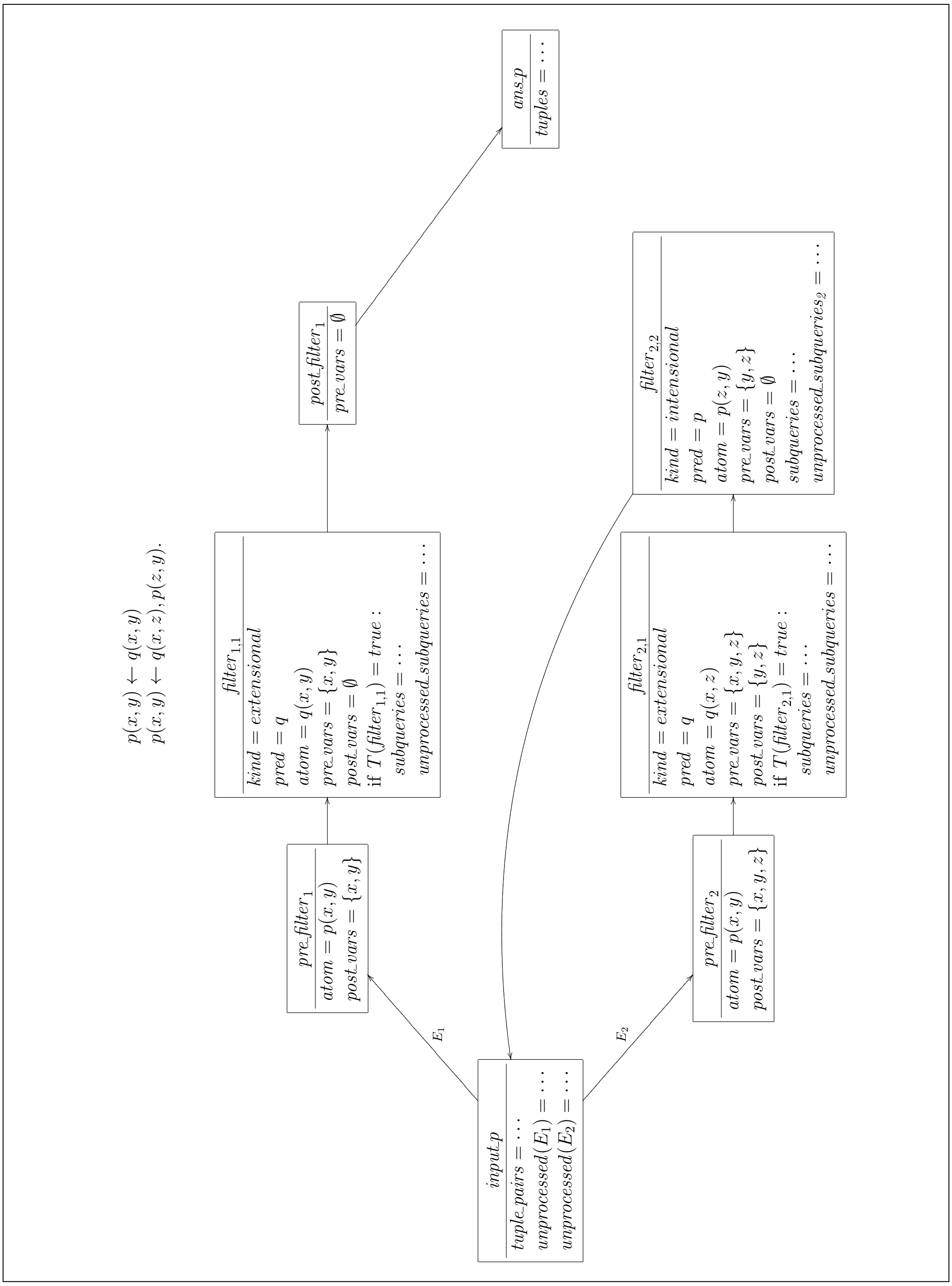


Fig. 4.2: The QSQN-TRE of the program given in Example 3.1 with $T(p) = \mathit{true}$.

– If $v = \mathit{filter}_{i,n_i}$, $\mathit{kind}(v) = \mathit{intensional}$, $\mathit{pred}(v) = p$, $T(p) = \mathit{true}$ and p is the predicate of A_i then transfer the tuple pair $(\bar{t}', \bar{t})$ such that $p(\bar{t}') = \mathit{atom}(v)\delta = B_{i,n_i}\delta$ through $(v, \mathit{input\_p})$ to add its fresh variant to $\mathit{tuple\_pairs}(\mathit{input\_p})$.
– If $v = \mathit{post\_filter}_i$ and p is the predicate of A_i then transfer the tuple $\bar{t}$ through $(\mathit{post\_filter}_i, \mathit{ans\_p})$ to add it to $\mathit{tuples}(\mathit{ans\_p})$.

Formally, in the same way as for the QSQN method, the processing of a subquery, an input/answer tuple or an input tuple pair in a QSQN-TRE is designed so that:

– every subquery or input/answer tuple or input tuple pair that is subsumed by another one or has a term-depth greater than a fixed bound l is ignored,
– the processing is divided into smaller steps which can be delayed at each node to maximize adjustability and allow various control strategies,
– the processing is done set-at-a-time (e.g., for all the unprocessed subqueries accumulated in a given node).

All of the related procedures and functions are listed in Appendix C. In particular, the procedure `transfer2`(D, u, v) (on pages 113-114) specifies the effects of transferring data D through an edge (u, v) of a QSQN-TRE. If v is of the form $\mathit{pre\_filter}_i$ or $\mathit{post\_filter}_i$ or ($v = \mathit{filter}_{i,j}$ and $\mathit{kind}(v) = \mathit{extensional}$ and $T(v) = \mathit{false}$) then the input D for v is processed immediately and an appropriate data Γ is produced and transferred through $(v, \mathit{succ}(v))$. Otherwise, the input D for v is not processed immediately, but accumulated into the contents of v in an appropriate way.

The function `active-edge`(u, v) (on page 28) returns *true* for an edge (u, v) if the data accumulated in u can be processed to produce some data to transfer through (u, v), and returns *false* otherwise. If `active-edge`(u, v) is *true* then the procedure `fire2`(u, v) (on page 112) processes the data accumulated in u that has not been processed before to transfer appropriate data through the edge (u, v). This procedure uses the procedure `transfer2`(D, u, v). Both the procedures `fire2`(u, v) and `transfer2`(D, u, v) use a parameter l as a term-depth bound for tuples and substitutions.

Algorithm 2 (on page 38) presents our QSQN-TRE evaluation method for Horn knowledge bases. It repeatedly selects an active edge and fires the operation for the edge. Such a selection is decided by the adopted control strategy, which can be arbitrary. If there is no tail-recursion to eliminate or $T(p) = \mathit{false}$ for every intensional predicate p, the QSQN-TRE method reduces to the QSQN evaluation method.

Example 4.2. The aim of this example is to illustrate how the QSQN-TRE method works in detail. It uses the logic program P, the extensional instance I and the query as in Example 3.3. The QSQN-TRE topological structure of the given program P is illustrated in Figure 4.3. For convenience, we name the edges of the net by E_i with $1 \leq i \leq 14$ as shown in this figure. We assume that $T(p) = true$, and $T(v) = false$ for each $v = \mathit{filter}_{i,j} \in V$ with $\mathit{kind}(v) = \mathit{extensional}$. We also assume that Algorithm 2 fires active edges in the order (E_1, E_3, E_4, E_7, E_4, E_7, E_4, E_7, E_4, E_8, E_{12}, E_{13}), which corresponds to the IDFS control strategy specified in Chapter 6.

We give below a trace of firing each "active edges" in the above list. The list of edges in the first row of the following steps denotes a call of the procedure `fire2` for

Algorithm 2: for evaluating a query $(P, q(\overline{x}))$ on an extensional instance I.

let (V, E, T) be a QSQN-TRE structure of P;
`// T can be chosen arbitrarily or appropriately`
set C so that $N = (V, E, T, C)$ is an empty QSQN-TRE of P;

let $\overline{x}'$ be a fresh variant of $\overline{x}$;
if $T(q) = \mathit{false}$ **then**
  $\mathit{tuples}(\mathit{input\_q}) := \{\overline{x}'\}$;
  foreach $(\mathit{input\_q}, v) \in E$ **do** $\mathit{unprocessed}(\mathit{input\_q}, v) := \{\overline{x}'\}$;
else
  $\mathit{tuple\_pairs}(\mathit{input\_q}) := \{(\overline{x}', \overline{x}')\}$;
  foreach $(\mathit{input\_q}, v) \in E$ **do** $\mathit{unprocessed}(\mathit{input\_q}, v) := \{(\overline{x}', \overline{x}')\}$;

while *there exists* $(u, v) \in E$ *such that* `active-edge`(u, v) *holds* **do**
  select $(u, v) \in E$ such that `active-edge`(u, v) holds;
  `// any strategy is acceptable for the above selection`
  `fire2`(u, v)

return $\mathit{tuples}(\mathit{ans\_q})$

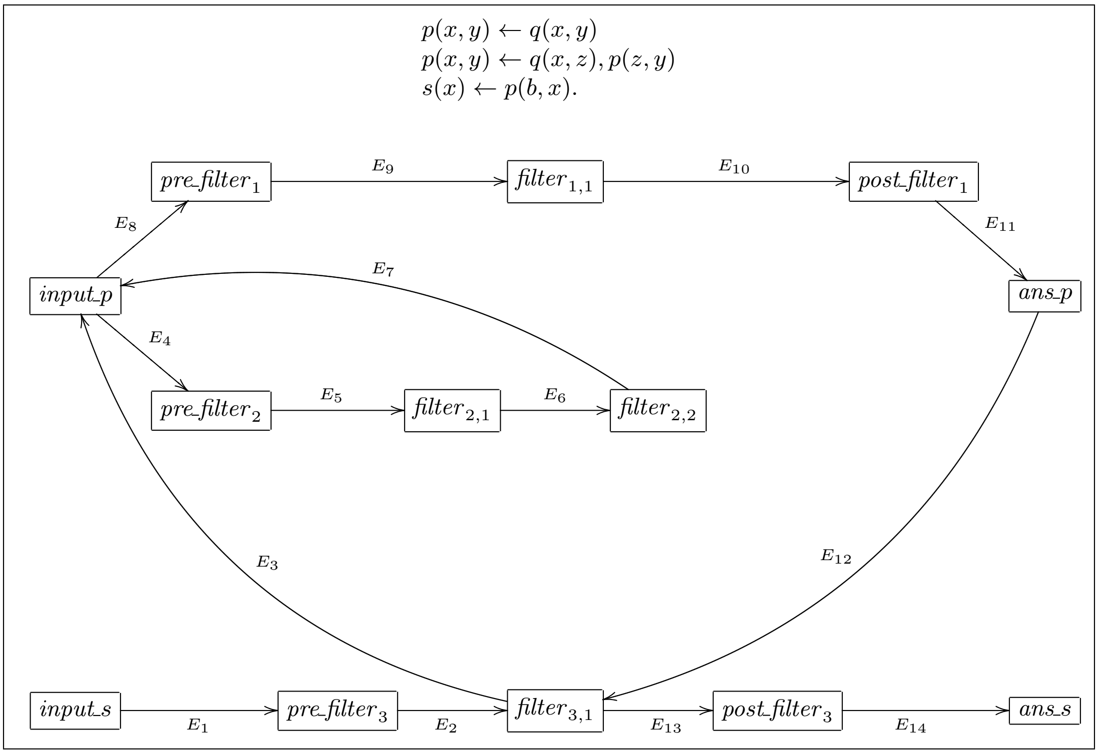
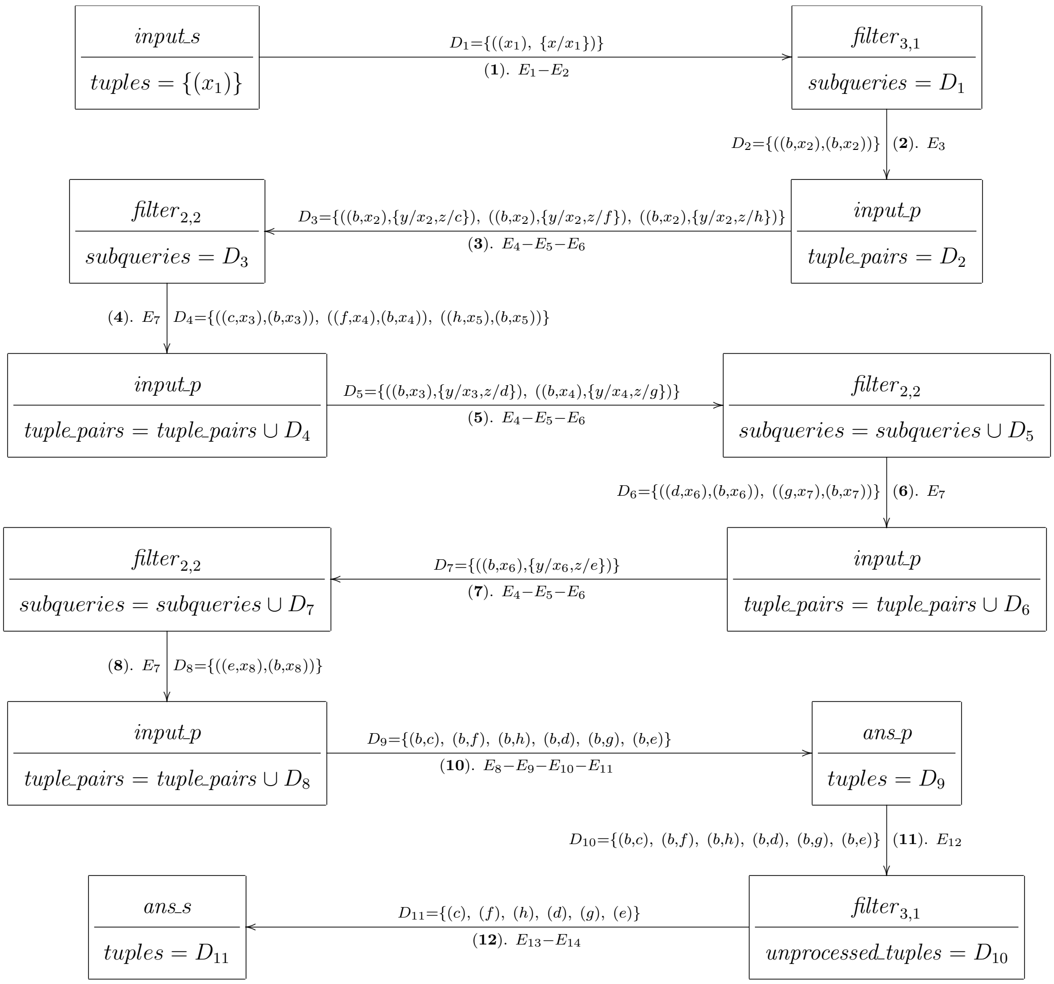


Fig. 4.3: The QSQN-TRE topological structure of the program given in Example 4.2.



input_s	*ans_s*	*input_p*	*ans_p*
x_1 **(0)**	c **(12)**	$((b,x_2),(b,x_2))$ **(2)**	(b,c) **(10)**
	f	$((c,x_3),(b,x_3))$ **(4)**	(b,f)
	h	$((f,x_4),(b,x_4))$	(b,h)
	d	$((h,x_5),(b,x_5))$	(b,d)
	g	$((d,x_6),(b,x_6))$ **(6)**	(b,g)
	e	$((g,x_7),(b,x_7))$	(b,e)
		$((e,x_8),(b,x_8))$ **(8)**	

Fig. 4.4: A view of tracing the execution of Algorithm 2 on the query given in Example 4.2.

the first edge in the list, which triggers transferring data through the subsequent edges of the list.

Algorithm 2 starts with an empty QSQN-TRE. It then adds a fresh variant (x_1) of (x) to the empty sets $tuples(input\_s)$ and $unprocessed(E_1)$. Next, it repeatedly selects and fires an active edge as follows (according to the order of the above list).

1. $\mathbf{E_1 - E_2}$

   After processing $unprocessed(E_1)$, the algorithm empties this set and transfers $\{(x_1)\}$ through the edge E_1. This produces $\{((x_1), \{x/x_1\})\}$, which is then transferred through the edge E_2 and added to the empty sets $subqueries(filter_{3,1})$, $unprocessed\_subqueries(filter_{3,1})$ and $unprocessed\_subqueries_2(filter_{3,1})$.

2. $\mathbf{E_3}$

   After processing $unprocessed\_subqueries_2(filter_{3,1})$, the algorithm empties this set, produces a pair $((b, x_1), (b, x_1))$, transfers its fresh variant $((b, x_2), (b, x_2))$ through E_3, and adds this variant to the empty sets $tuple\_pairs(input\_p)$, $unprocessed(E_4)$ and $unprocessed(E_8)$.

3. $\mathbf{E_4 - E_5 - E_6}$

   After processing $unprocessed(E_4)$, the algorithm empties this set and transfers $\{((b, x_2), (b, x_2))\}$ through the edge E_4. This produces $\{((b, x_2), \{x/b, y/x_2\})\}$, which is then transferred through the edge E_5, producing $\{((b, x_2), \{y/x_2, z/c\}), ((b, x_2), \{y/x_2, z/f\}), ((b, x_2), \{y/x_2, z/h\})\}$, which in turn is then transferred through the edge E_6 and added to the empty sets $subqueries(filter_{2,2})$ and $unprocessed\_subqueries_2(filter_{2,2})$.

4. $\mathbf{E_7}$

   After processing $unprocessed\_subqueries_2(filter_{2,2})$, the algorithm empties this set, produces a set of pairs $\{((c, x_2), (b, x_2)), ((f, x_2), (b, x_2)), ((h, x_2), (b, x_2))\}$, transfers its fresh variant $\{((c, x_3), (b, x_3)), ((f, x_4), (b, x_4)), ((h, x_5), (b, x_5))\}$ through the edge E_7, and adds this variant to the sets $tuple\_pairs(input\_p)$, $unprocessed(E_4)$ and $unprocessed(E_8)$. After these steps, we have:

   - $unprocessed(E_8) = tuple\_pairs(input\_p) =$

     $\{((b, x_2), (b, x_2)), ((c, x_3), (b, x_3)), ((f, x_4), (b, x_4)), ((h, x_5), (b, x_5))\}$,

   - $unprocessed(E_4) = \{((c, x_3), (b, x_3)), ((f, x_4), (b, x_4)), ((h, x_5), (b, x_5))\}$.

5. $\mathbf{E_4 - E_5 - E_6}$

   After processing $unprocessed(E_4)$, the algorithm empties this set and transfers $\{((c, x_3), (b, x_3)), ((f, x_4), (b, x_4)), ((h, x_5), (b, x_5))\}$ through the edge E_4. This produces $\{((b, x_3), \{x/c, y/x_3\}), ((b, x_4), \{x/f, y/x_4\}), ((b, x_5), \{x/h, y/x_5\})\}$, which is then transferred through the edge E_5, producing $\{((b, x_3), \{y/x_3, z/d\}), ((b, x_4), \{y/x_4, z/g\})\}$, which in turn is then transferred through the edge E_6 and added to $subqueries(filter_{2,2})$ and $unprocessed\_subqueries_2(filter_{2,2})$. After these steps, we have:

   - $subqueries(filter_{2,2}) = \{((b, x_2), \{y/x_2, z/c\}), ((b, x_2), \{y/x_2, z/f\}),$

     $((b, x_2), \{y/x_2, z/h\}), ((b, x_3), \{y/x_3, z/d\}), ((b, x_4), \{y/x_4, z/g\})\}$,

- *unprocessed_subqueries$_2$*(*filter*$_{2,2}$) $= \{((b, x_3), \{y/x_3, z/d\}),\ ((b, x_4), \{y/x_4, z/g\})\}$.

6. $\mathbf{E_7}$

   After processing *unprocessed_subqueries$_2$*(*filter*$_{2,2}$), the algorithm empties this set, produces a set of pairs $\{((d, x_3), (b, x_3)),\ ((g, x_4), (b, x_4))\}$, transfers its fresh variant $\{((d, x_6), (b, x_6)),\ ((g, x_7), (b, x_7))\}$ through the edge E_7, and adds this variant to the sets *tuple_pairs*(*input_p*), *unprocessed*(E_4) and *unprocessed*(E_8). After these steps, we have:

   - *unprocessed*(E_8) = *tuple_pairs*(*input_p*) $= \{((b, x_2), (b, x_2)),\ ((c, x_3), (b, x_3)),$
   $((f, x_4), (b, x_4)),\ ((h, x_5), (b, x_5)),\ ((d, x_6), (b, x_6)),\ ((g, x_7), (b, x_7))\}$,
   - *unprocessed*(E_4) $= \{((d, x_6), (b, x_6)),\ ((g, x_7), (b, x_7))\}$.

7. $\mathbf{E_4 - E_5 - E_6}$

   After processing *unprocessed*(E_4), the algorithm empties this set and transfers $\{((d, x_6), (b, x_6)),\ ((g, x_7), (b, x_7))\}$ through the edge E_4. This produces $\{((b, x_6), \{x/d, y/x_6\}),\ ((b, x_7), \{x/g, y/x_7\})\}$, which is then transferred through the edge E_5, producing $\{((b, x_6), \{y/x_6, z/e\})\}$, which in turn is then transferred through the edge E_6 and added to *subqueries*(*filter*$_{2,2}$) and *unprocessed_subqueries$_2$*(*filter*$_{2,2}$). After these steps, we have:

   - *subqueries*(*filter*$_{2,2}$) $= \{((b, x_2), \{y/x_2, z/c\}),\ ((b, x_2), \{y/x_2, z/f\}),$
   $((b, x_2), \{y/x_2, z/h\}),\ ((b, x_3), \{y/x_3, z/d\}),$
   $((b, x_4), \{y/x_4, z/g\}),\ ((b, x_6), \{y/x_6, z/e\})\}$,
   - *unprocessed_subqueries$_2$*(*filter*$_{2,2}$) $= \{((b, x_6), \{y/x_6, z/e\})\}$.

8. $\mathbf{E_7}$

   After processing *unprocessed_subqueries$_2$*(*filter*$_{2,2}$), the algorithm empties this set, produces a pair $\{((e, x_6), (b, x_6))\}$, transfers its fresh variant $\{((e, x_8), (b, x_8))\}$ through the edge E_7, and adds this variant to the sets *tuple_pairs*(*input_p*), *unprocessed*(E_4) and *unprocessed*(E_8). After these steps, we have:

   - *unprocessed*(E_8) = *tuple_pairs*(*input_p*) $= \{((b, x_2), (b, x_2)),\ ((c, x_3), (b, x_3)),$
   $((f, x_4), (b, x_4)),\ ((h, x_5), (b, x_5)),\ ((d, x_6), (b, x_6)),\ ((g, x_7), (b, x_7)),\ ((e, x_8), (b, x_8))\}$,
   - *unprocessed*(E_4) $= \{((e, x_8), (b, x_8))\}$.

9. $\mathbf{E_4 - E_5 - E_6}$

   After processing *unprocessed*(E_4), the algorithm empties this set and transfers $\{((e, x_8), (b, x_8))\}$ through the edge E_4. This produces $\{((b, x_8), \{x/e, y/x_8\})\}$, which is then transferred through the edge E_5, producing nothing.

10. $\mathbf{E_8 - E_9 - E_{10} - E_{11}}$

    After processing *unprocessed*(E_8), the algorithm empties this set and transfers the set of pairs $\{((b, x_2), (b, x_2)),\ ((c, x_3), (b, x_3)),\ ((f, x_4), (b, x_4)),\ ((h, x_5), (b, x_5)),$ $((d, x_6), (b, x_6)),\ ((g, x_7), (b, x_7)),\ ((e, x_8), (b, x_8))\}$ through the edge E_8. This produces a set of subqueries $\{((b, x_2), \{x/b, y/x_2\}),\ ((b, x_3), \{x/c, y/x_3\}),$ $((b, x_4), \{x/f, y/x_4\}),\ ((b, x_5), \{x/h, y/x_5\}),\ ((b, x_6), \{x/d, y/x_6\}),$

$((b, x_7), \{x/g, y/x_7\})$, $((b, x_8), \{x/e, y/x_8\})\}$, which is then transferred through the edge E_9, producing $\{((b, c), \varepsilon)$, $((b, f), \varepsilon)$, $((b, h), \varepsilon)$, $((b, d), \varepsilon)$, $((b, g), \varepsilon)$, $((b, e), \varepsilon)\}$, which in turn is then transferred through the edge E_{10}, producing $\{(b, c)$, (b, f), (b, h), (b, d), (b, g), $(b, e)\}$, which in turn is then transferred through the edge E_{11} and added to the empty sets *tuples*(*ans_p*) and *unprocessed*(E_{12}).

11. $\mathbf{E_{12}}$

    After processing *unprocessed*(E_{12}), the algorithm empties this set and transfers $\{(b, c)$, (b, f), (b, h), (b, d), (b, g), $(b, e)\}$ through the edge E_{12} and adds these tuples to the empty set *unprocessed_tuples*($filter_{3,1}$).

12. $\mathbf{E_{13} - E_{14}}$

    After processing the sets *unprocessed_subqueries*($filter_{3,1}$) and *unprocessed_tuples*($filter_{3,1}$), the algorithm empties these sets and transfers $\{((c), \varepsilon)$, $((f), \varepsilon)$, $((h), \varepsilon)$, $((d), \varepsilon)$, $((g), \varepsilon)$, $((e), \varepsilon)\}$ through the edge E_{13}. This produces $\{(c), (f), (h), (d), (g), (e)\}$, which is then transferred through the edge E_{14} and added to the empty set *tuples*(*ans_s*).

At this point, no edge is active (in particular, all the attributes *unprocessed*, *unprocessed_subqueries*, *unprocessed_subqueries*$_2$ and *unprocessed_tuples* of the nodes in the net are empty sets). The algorithm terminates and returns the set of results in *tuples*(*ans_s*) $= \{(c), (f), (h), (d), (g), (e)\}$.

Figure 4.4 (on page 39) shows an intuitive view of this trace. In this figure, D_i $(1 \leq i \leq 11)$ presents the data transferred through the last edge in the corresponding list of edges. The table summarizes the steps at which the data (i.e., a set of tuples or tuple pairs) were added to *input_s*, *ans_s*, *input_p*, *ans_p*, respectively. ∎

4.1.2 Soundness and Completeness

The following lemmas state a property of Algorithm 2. The proof of Lemma 4.1 is straightforward.

Lemma 4.1. *Consider a run of Algorithm 2 (using parameter l) on a query $(P, q(\overline{x}))$ and an extensional instance I. Let $v = filter_{i,j}$ for some $1 \leq i \leq m$ and $1 \leq j \leq n_i$ such that if $j = n_i$ then φ_i is not tail-recursive or $T(p) = false$ or p is not the predicate of A_i. Let $w = succ(v)$ and let $u = filter_{i,j-1}$ if $j > 1$, and $u = pre\_filter_i$ otherwise. Suppose that a subquery $(\overline{s}', \delta')$ was transferred through (v, w) at some step k. Then, there exists a subquery $(\overline{s}, \delta)$ which was transferred through (u, v) at some earlier step $h < k$ with the property that:*

- *if $kind(v) = extensional$ and $pred(v) = p$ then there exists $\overline{t}' \in I(p)$ such that $atom(v)\delta$ is unifiable with a fresh variant of $p(\overline{t}')$ by an mgu γ, $\overline{s}' = \overline{s}\gamma$ and $\delta' = (\delta\gamma)_{|post\_vars(v)}$,*
- *if $kind(v) = intensional$ and $pred(v) = p$ then there was $\overline{t}' \in tuples(ans\_p)$ at step k such that $atom(v)\delta$ is unifiable with a fresh variant of $p(\overline{t}')$ by an mgu γ, $\overline{s}' = \overline{s}\gamma$ and $\delta' = (\delta\gamma)_{|post\_vars(v)}$.* ∎

Lemma 4.2 (Soundness). *Consider a run of Algorithm 2 (using parameter l) on a query $(P, q(\overline{x}))$ and an extensional instance I. For every intensional predicate p of P,* *(a) if $T(p) = true$ then, for every pair $(\bar{t}, \bar{t}') \in tuple\_pairs(input\_p)$ and every substitution θ, if $P \cup I \models \forall(p(\bar{t})\theta)$ then $P \cup I \models \forall(p(\bar{t}')\theta)$,*

(b) every computed answer $\bar{s} \in tuples(ans\_p)$ is a correct answer in the sense that $P \cup I \models \forall(p(\bar{s}))$.

Proof. We prove this lemma by induction on the number of the step at which either the pair $(\bar{t}, \bar{t}')$ was added to $tuple\_pairs(input\_p)$ or the tuple $\bar{s}$ was added to $tuples(ans\_p)$. Consider the assertion (a) first and assume that $T(p) = true$.

- Suppose $(\bar{t}, \bar{t}')$ was added to $tuple\_pairs(input\_p)$ with the property that $\bar{t} = \bar{t}'$, which was performed by the following cases:
  - at the beginning of processing the query (Step 8 of the Algorithm 2),
  - when $kind(filter_{i,j}) = intensional$, $p = pred(filter_{i,j})$, $T(p) = true$ and ($j < n_i$ or p is not predicate of A_i) (Step 8 of the procedure `compute-gamma` (on page 111)).

  Clearly, in these cases, if $P \cup I \models \forall(p(\bar{t})\theta)$ then $P \cup I \models \forall(p(\bar{t}')\theta)$ for every substitution θ.
- We now consider the case when A_i and B_{i,n_i} have the same intensional predicate p with $T(p) = true$. Suppose $(\bar{t}, \bar{t}')$ was added to $tuple\_pairs(input\_p)$ as the result of transferring data through the edge $(filter_{i,n_i}, input\_p)$. Thus, there was a subquery $(\bar{t}'_{n_i}, \delta_{n_i}) \in unprocessed\_subqueries_2(filter_{i,n_i})$ such that $p(\bar{t}_{n_i}) = B_{i,n_i}\delta_{n_i}$ and $(\bar{t}, \bar{t}')$ is a fresh variant of $(\bar{t}_{n_i}, \bar{t}'_{n_i})$. Let $v_0 = pre\_filter_i$ and $v_j = filter_{i,j}$ for $1 \leq j \leq n_i$. The subquery $(\bar{t}'_{n_i}, \delta_{n_i})$ was added to $unprocessed\_subqueries_2(filter_{i,n_i})$ as the result of transferring a subquery $(\bar{t}'_{n_i-1}, \delta_{n_i-1})$ through the edge $(v_{n_i-1}, filter_{i,n_i})$ with the properties that $\bar{t}'_{n_i} = \bar{t}'_{n_i-1}$ and $\delta_{n_i} = \delta_{n_i-1}$ (Step 18 of the procedure `transfer`2). By Lemma 4.1, for each j from $n_i - 1$ to 1, there exists a subquery $(\bar{t}'_{j-1}, \delta_{j-1})$ transferred through (v_{j-1}, v_j) such that:

$$\text{if } kind(v_j) = extensional \text{ and } pred(v_j) = p_j \text{ then there exists } \bar{t}''_j \in I(p_j) \text{ such that } atom(v_j)\delta_{j-1} \text{ is unifiable with a fresh variant of } p_j(\bar{t}''_j) \text{ by an mgu } \gamma_j,\ \bar{t}'_j = \bar{t}'_{j-1}\gamma_j \text{ and } \delta_j = (\delta_{j-1}\gamma_j)_{|post\_vars(v_j)}, \tag{4.1}$$

$$\text{if } kind(v_j) = intensional \text{ and } pred(v_j) = p_j \text{ then there exists } \bar{t}''_j \in tuples(ans\_p_j) \text{ such that } atom(v_j)\delta_{j-1} \text{ is unifiable with a fresh variant of } p_j(\bar{t}''_j) \text{ by an mgu } \gamma_j,\ \bar{t}'_j = \bar{t}'_{j-1}\gamma_j \text{ and } \delta_j = (\delta_{j-1}\gamma_j)_{|post\_vars(v_j)}. \tag{4.2}$$

Additionally, there must exist a pair $(\bar{t}_\diamond, \bar{t}'_\diamond)$ which was added to $tuple\_pairs(input\_p)$ in an earlier step such that $\delta_0 = mgu(A_i, p(\bar{t}_\diamond))$ and $\bar{t}'_0 = \bar{t}'_\diamond\delta_0$. By the inductive assumption for (a), we have that, for every substitution θ, if $P \cup I \models \forall(p(\bar{t}_\diamond)\theta)$ then $P \cup I \models \forall(p(\bar{t}'_\diamond)\theta)$.

We prove by an inner induction on $1 \leq j \leq n_i$ that, for every substitution θ:

$$\text{if } P \cup I \models \forall((B_{i,j}, \ldots, B_{i,n_i})\delta_{j-1}\theta) \text{ then } P \cup I \models \forall(p(\bar{t}'_{j-1})\theta). \tag{4.3}$$

Base case (j = 1): Assume that $P \cup I \models \forall((B_{i,1}, \ldots, B_{i,n_i})\delta_0\theta)$. Since $P \cup I \models \forall(\varphi_i)$, we have $P \cup I \models \forall((B_{i,1}, \ldots, B_{i,n_i} \to A_i)\delta_0\theta)$. It follows that $P \cup I \models \forall(A_i\delta_0\theta)$. Since

$A_i\delta_0 = p(\bar{t}_\diamond)\delta_0$, we have $P \cup I \models \forall(p(\bar{t}_\diamond)\delta_0\theta)$. This implies that $P \cup I \models \forall(p(\bar{t}'_\diamond)\delta_0\theta)$. Since $\bar{t}'_0 = \bar{t}'_\diamond\delta_0$, we have that $P \cup I \models \forall(p(\bar{t}'_0)\theta)$.

Induction step: Suppose the induction hypothesis holds for $j < n_i$, i.e.

$$\text{for every } \theta',\ \text{if } P \cup I \models \forall((B_{i,j}, \ldots, B_{i,n_i})\delta_{j-1}\theta') \text{ then } P \cup I \models \forall(p(\bar{t}'_{j-1})\theta'). \tag{4.4}$$

We show that it also holds for $j+1$, i.e.,

$$\text{for every } \theta'',\ \text{if } P \cup I \models \forall((B_{i,j+1}, \ldots, B_{i,n_i})\delta_j\theta'') \text{ then } P \cup I \models \forall(p(\bar{t}'_j)\theta''). \tag{4.5}$$

Suppose

$$P \cup I \models \forall((B_{i,j+1}, \ldots, B_{i,n_i})\delta_j\theta''). \tag{4.6}$$

Take $\theta' = \gamma_j\theta''$.

- Consider the case $\mathit{kind}(v_j) = \mathit{extensional}$ and let $p_j = \mathit{pred}(v_j)$. By (4.1), there exists a fresh variant $\bar{t}^*_j$ of some $\bar{t}''_j \in I(p_j)$ such that $\gamma_j = \mathit{mgu}(B_{i,j}\delta_{j-1}, p_j(\bar{t}^*_j))$, $\bar{t}'_j = \bar{t}'_{j-1}\gamma_j$ and $\delta_j = (\delta_{j-1}\gamma_j)_{|\mathit{post\_vars}(v_j)}$. We have that $P \cup I \models \forall(p_j(\bar{t}''_j))$, hence $P \cup I \models \forall(p_j(\bar{t}^*_j)\gamma_j)$, which means $P \cup I \models \forall(B_{i,j}\delta_{j-1}\gamma_j)$. Hence $P \cup I \models \forall(B_{i,j}\delta_{j-1}\gamma_j\theta'')$, which implies

$$P \cup I \models \forall(B_{i,j}\delta_{j-1}\theta'). \tag{4.7}$$

  Since $\delta_j = (\delta_{j-1}\gamma_j)_{|\mathit{post\_vars}(v_j)}$ and $\theta' = \gamma_j\theta''$, we have that

$$(B_{i,j+1}, \ldots, B_{i,n_i})\delta_j\theta'' = (B_{i,j+1}, \ldots, B_{i,n_i})\delta_{j-1}\theta'.$$

  This together with (4.6), (4.7) and (4.4) implies $P \cup I \models \forall(p(\bar{t}'_{j-1})\theta')$. Since $\bar{t}'_{j-1}\theta' = \bar{t}'_{j-1}\gamma_j\theta'' = \bar{t}'_j\theta''$, it follows that $P \cup I \models \forall(p(\bar{t}'_j)\theta'')$, which completes the proof of (4.5) for the case $\mathit{kind}(v_j) = \mathit{extensional}$.

- Consider the case $\mathit{kind}(v_j) = \mathit{intensional}$ and let $p_j = \mathit{pred}(v_j)$. By (4.2), there exists a fresh variant $\bar{t}^*_j$ of some $\bar{t}''_j \in \mathit{tuples}(\mathit{ans\_p_j})$ such that $\gamma_j = \mathit{mgu}(B_{i,j}\delta_{j-1}, p_j(\bar{t}^*_j))$, $\bar{t}'_j = \bar{t}'_{j-1}\gamma_j$ and $\delta_j = (\delta_{j-1}\gamma_j)_{|\mathit{post\_vars}(v_j)}$. By the inductive assumption of the outer induction for (b), we have $P \cup I \models \forall(p_j(\bar{t}''_j))$, hence $P \cup I \models \forall(p_j(\bar{t}^*_j)\gamma_j)$, which means $P \cup I \models \forall(B_{i,j}\delta_{j-1}\gamma_j)$. Analogously as for the above case, we can derive that $P \cup I \models \forall(p(\bar{t}'_j)\theta'')$, which completes the proof of (4.5) for the case $\mathit{kind}(v_j) = \mathit{intensional}$ and the proof of (4.3).

Recall that $p(\bar{t}_{n_i}) = B_{i,n_i}\delta_{n_i}$, $\bar{t}'_{n_i} = \bar{t}'_{n_i-1}$ and $\delta_{n_i} = \delta_{n_i-1}$. By (4.3), when $j = n_i$, we have that, for every substitution θ, if $P \cup I \models \forall(B_{i,n_i}\delta_{n_i-1}\theta)$ then $P \cup I \models \forall(p(\bar{t}'_{n_i-1})\theta)$. Hence, if $P \cup I \models \forall(p(\bar{t}_{n_i})\theta)$ then $P \cup I \models \forall(p(\bar{t}'_{n_i})\theta)$. Since $(\bar{t}, \bar{t}')$ is a fresh variant of $(\bar{t}_{n_i}, \bar{t}'_{n_i})$, it follows that, for every substitution θ, if $P \cup I \models \forall(p(\bar{t})\theta)$ then $P \cup I \models \forall(p(\bar{t}')\theta)$. This completes the proof of (a).

Now, consider the assertion (b). Suppose that $\overline{s}$ was added to $tuples(ans\_p)$ as the result of transferring $\overline{s}$ through the edge $(post\_filter_i, ans\_p)$, which was triggered by the transfer of $(\overline{s}, \varepsilon)$ through the edge $(pre\_filter_i, post\_filter_i)$ if $n_i = 0$ or $(filter_{i,n_i}, post\_filter_i)$ otherwise. Let $\overline{t}'_{n_i} = \overline{s}$ and $\delta_{n_i} = \varepsilon$. Let $v_0 = pre\_filter_i$ and $v_j = filter_{i,j}$ for $1 \leq j \leq n_i$. By Lemma 4.1, for each j from n_i to 1, there exists a subquery $(\overline{t}'_{j-1}, \delta_{j-1})$ transferred through (v_{j-1}, v_j) such that:

$$\text{if } kind(v_j) = extensional \text{ and } pred(v_j) = p_j \text{ then there exists } \overline{t}''_j \in I(p_j) \text{ such that } atom(v_j)\delta_{j-1} \text{ is unifiable with a fresh variant of } p_j(\overline{t}''_j) \text{ by an mgu } \gamma_j,\ \overline{t}'_j = \overline{t}'_{j-1}\gamma_j \text{ and } \delta_j = (\delta_{j-1}\gamma_j)_{|post\_vars(v_j)}, \tag{4.8}$$

$$\text{if } kind(v_j) = intensional \text{ and } pred(v_j) = p_j \text{ then there exists } \overline{t}''_j \in tuples(ans\_p_j) \text{ such that } atom(v_j)\delta_{j-1} \text{ is unifiable with a fresh variant of } p_j(\overline{t}''_j) \text{ by an mgu } \gamma_j,\ \overline{t}'_j = \overline{t}'_{j-1}\gamma_j \text{ and } \delta_j = (\delta_{j-1}\gamma_j)_{|post\_vars(v_j)}. \tag{4.9}$$

Consider the case $T(p) = true$. By an analogous proof as for (4.3), we have that, for $1 \leq j \leq n_i + 1$:

$$\text{for every substitution } \theta, \text{ if } P \cup I \models \forall((B_{i,j}, \ldots, B_{i,n_i})\delta_{j-1}\theta) \text{ then } P \cup I \models \forall(p(\overline{t}'_{j-1})\theta). \tag{4.10}$$

Consider the case $T(p) = false$. There must exist a tuple $\overline{t}'_\diamond$ which was added to $tuples(input\_p)$ in an earlier step such that $\delta_0 = mgu(A_i, p(\overline{t}'_\diamond))$ and $\overline{t}'_0 = \overline{t}'_\diamond\delta_0$. We have that $A_i\delta_0 = p(\overline{t}'_0)$. We now prove that (4.10) also holds for the case $T(p) = false$ by an inner induction on $1 \leq j \leq n_i + 1$.

Base case ($j = 1$): Assume that $P \cup I \models \forall((B_{i,1}, \ldots, B_{i,n_i})\delta_0\theta)$. Since $P \cup I \models \forall(\varphi_i)$, we have $P \cup I \models \forall((B_{i,1} \wedge \ldots \wedge B_{i,n_i} \rightarrow A_i)\delta_0\theta)$. It follows that $P \cup I \models \forall(A_i\delta_0\theta)$, which means $P \cup I \models \forall(p(\overline{t}'_0)\theta)$.

The induction step is similar to the one given for (4.3).

By (4.10), when $j = n_i + 1$ and $\theta = \varepsilon$, we have that $P \cup I \models \forall(p(\overline{t}'_{n_i}))$, which means $P \cup I \models \forall(p(\overline{s}))$, which completes the proof of (b) and also the proof of this lemma. ∎

We need the following lemma for the completeness theorem. We assume that the sets of fresh variables used for renaming variables of input program clauses in SLD-refutations and in Algorithm 2 are disjoint.

Lemma 4.3. *After a run of Algorithm 2 (using parameter l) on a query $(P, q(\overline{x}))$ and an extensional instance I, for every intensional predicate r of P, for every SLD-refutation of $P \cup I \cup \{\leftarrow r(\overline{s})\}$ that uses the leftmost selection function, does not contain any goal with term-depth greater than l and has a computed answer θ with the term-depth of $\overline{s}\theta$ not greater than l,*

- *if $T(r) = false$ and $\overline{s} \in tuples(input\_r)$ then there exists $\overline{s}'' \in tuples(ans\_r)$ such that $\overline{s}\theta$ is an instance of a variant of $\overline{s}''$,*
- *if $T(r) = true$ and $(\overline{s}, \overline{s}') \in tuple\_pairs(input\_r)$, then there exists $\overline{s}'' \in tuples(ans\_r)$ such that $\overline{s}'\theta$ is an instance of a variant of $\overline{s}''$.*

Proof. We prove this lemma by induction on the length of the mentioned SLD-refutation. We give bellow the proof for the case $T(r) = \mathit{true}$. The case $T(r) = \mathit{false}$ is simpler and the proof for it is given in Appendix B.

Suppose that $T(r) = \mathit{true}$ and the first step of the refutation of $P \cup I \cup \{\leftarrow r(\overline{s})\}$ uses an input program clause $\varphi'_i = (A'_i \leftarrow B'_{i,1}, \ldots, B'_{i,n_i})$, which is a variant of a clause $\varphi_i = (A_i \leftarrow B_{i,1}, \ldots, B_{i,n_i})$ of P, resulting in the resolvent $\leftarrow (B'_{i,1}, \ldots, B'_{i,n_i})\theta_1$. Let $\theta_1, \ldots, \theta_y$ be the sequence of mgu's used in the refutation. By the definition of computed answers, we have $\theta = (\theta_1 \ldots \theta_y)_{|\mathit{Vars}(\overline{s})}$. Observe that only fresh variants of tuple pairs were added to $\mathit{tuple\_pairs}(\mathit{input\_r})$. Recall the assumption that the set of variables used for renaming variables in Algorithm 2 is disjoint with the set of variables used for renaming variables in SLD-derivations. Hence, $(\mathit{Vars}(\overline{s}')\setminus\mathit{Vars}(\overline{s})) \cap \mathit{Vars}(\theta_1 \ldots \theta_y) = \emptyset$. It follows that

$$\overline{s}'\theta_1 \ldots \theta_y = \overline{s}'((\theta_1 \ldots \theta_y)_{|\mathit{Vars}(\overline{s})}) = \overline{s}'\theta. \tag{4.11}$$

Let ϱ be a renaming substitution such that $\varphi'_i = \varphi_i\varrho$. Thus, $B'_{i,j} = B_{i,j}\varrho$ for $1 \leq j \leq n_i$. We can assume that ϱ does not use any variable occurring in $\overline{s}$ and $\overline{s}'$. Thus,

$$\overline{s} = \overline{s}\varrho, \tag{4.12}$$

and

$$\overline{s}' = \overline{s}'\varrho. \tag{4.13}$$

Let $k_1 = 2$, $k_{n_i+1} = y + 1$ and suppose that, for $1 \leq j \leq n_i$,

the fragment for processing $\leftarrow B'_{i,j}\theta_1 \ldots \theta_{k_j-1}$ of the refutation of $P \cup I \cup \{\leftarrow r(\overline{s})\}$ uses mgu's $\theta_{k_j}, \ldots, \theta_{k_{j+1}-1}$. (4.14)

Since $\theta_1 = \mathit{mgu}(r(\overline{s}), A'_i)$ and $A'_i = A_i\varrho$ and by (4.12), it follows that $r(\overline{s})\varrho\theta_1 = r(\overline{s})\theta_1 = A'_i\theta_1 = A_i\varrho\theta_1$ and hence $\varrho\theta_1$ is a unifier for $r(\overline{s})$ and A_i. Let γ_0 be an mgu Algorithm 2 used to unify $r(\overline{s})$ and A_i when processing $(\overline{s}, \overline{s}')$ for the edge $(\mathit{input\_r}, \mathit{pre\_filter}_i)$. Hence, there exists a substitution η_0 such that

$$\varrho\theta_1 = \gamma_0\eta_0. \tag{4.15}$$

Let $\overline{s}'_0 = \overline{s}'\gamma_0$ and $\delta_0 = (\gamma_0)_{|\mathit{post\_vars}(\mathit{pre\_filter}_i)}$.

Consider the base case, which occurs when $n_i = 0$ and the SLD-refutation has the length one. By (4.13) and (4.15), we have that

$$\overline{s}'\theta_1 = \overline{s}'\varrho\theta_1 = \overline{s}'\gamma_0\eta_0 = \overline{s}'_0\eta_0. \tag{4.16}$$

Thus, $\overline{s}'\theta_1$ is an instance of $\overline{s}'_0$. Since $\mathit{post\_vars}(\mathit{pre\_filter}_i) = \emptyset$, the subquery $(\overline{s}'_0, \varepsilon)$ was transferred through the edge $(\mathit{pre\_filter}_i, \mathit{post\_filter}_i)$. Hence, $\mathit{tuples}(\mathit{ans\_r})$ contains $\overline{s}''$ such that $\overline{s}'_0$ is an instance of a fresh variant of $\overline{s}''$. Since $\overline{s}'\theta = \overline{s}'\theta_1$, it follows that, $\overline{s}'\theta$ is an instance of a variant of $\overline{s}''$.

Let us consider the induction step. We have that $n_i \geq 1$. We will refer to the data structures used by Algorithm 2. We first prove the following remark:

Remark 4.2. *Let $v = \mathit{filter}_{i,j}$ for some $1 \le i \le m$ and $1 \le j < n_i$ if φ_i is tail-recursive and $1 \le j \le n_i$ otherwise. Let $u = \mathit{filter}_{i,j-1}$ if $j > 1$, and $u = \mathit{pre\_filter}_i$ otherwise. If $(\overline{s}'_{j-1}, \delta_{j-1})$ is a subquery transferred through (u, v) at some step and there exists a substitution η such that*

$$(\overline{s}', (B_{i,j}, \ldots, B_{i,n_i}))\varrho\theta_1 \ldots \theta_{k_j-1} = (\overline{s}'_{j-1}, (B_{i,j}, \ldots, B_{i,n_i})\delta_{j-1})\eta, \tag{4.17}$$

then there exists a subquery $(\overline{s}'_j, \delta_j)$ transferred through $(v, \mathit{succ}(v))$ at some step and a substitution η' such that

$$(\overline{s}', (B_{i,j+1}, \ldots, B_{i,n_i}))\varrho\theta_1 \ldots \theta_{k_{j+1}-1} = (\overline{s}'_j, (B_{i,j+1}, \ldots, B_{i,n_i})\delta_j)\eta'. \tag{4.18}$$

Suppose the premises of this remark hold. Without loss of generality we assume that:

$$\text{if } (\mathit{kind}(v) = \mathit{extensional} \text{ and } T(v) = \mathit{true}) \text{ or } \mathit{kind}(v) = \mathit{intensional} \text{ then the subquery } (\overline{s}'_{j-1}, \delta_{j-1}) \text{ was added to } \mathit{subqueries}(v). \tag{4.19}$$

Since $B'_{i,j} = B_{i,j}\varrho$ and (4.17), we have that:

$$(\leftarrow B'_{i,j}\theta_1 \ldots \theta_{k_j-1}) = (\leftarrow B_{i,j}\varrho\theta_1 \ldots \theta_{k_j-1}) = (\leftarrow B_{i,j}\delta_{j-1}\eta). \tag{4.20}$$

Since the term-depth of $B_{i,j}\delta_{j-1}\eta = B'_{i,j}\theta_1 \ldots \theta_{k_j-1}$ is not greater than l, the term-depth of $B_{i,j}\delta_{j-1}$ is also not greater than l. By (4.14), (4.20) and Lifting Lemma 2.2, we have that

> there exists a refutation of $P \cup I \cup \{\leftarrow B_{i,j}\delta_{j-1}\}$ using the leftmost selection function and mgu's $\theta'_{k_j}, \ldots, \theta'_{k_{j+1}-1}$ such that the term-depths of goals are not greater than l and $\eta\theta_{k_j} \ldots \theta_{k_{j+1}-1} = \theta'_{k_j} \ldots \theta'_{k_{j+1}-1}\mu$ for some substitution μ. (4.21)

Consider the case when the predicate $p = \mathit{pred}(v)$ of $B_{i,j}$ is an extensional predicate. Thus,

$$k_{j+1} = k_j + 1 \tag{4.22}$$

and

$$B_{i,j}\delta_{j-1}\theta'_{k_j} = p(\overline{t}')\sigma\theta'_{k_j} \tag{4.23}$$

where $p(\overline{t}')\sigma$ is the input program clause used for resolving $\leftarrow B_{i,j}\delta_{j-1}$, with $\overline{t}' \in I(p)$ and σ being a renaming substitution. Regarding the transfer of the subquery $(\overline{s}'_{j-1}, \delta_{j-1})$ through (u, v), under the assumption (4.19), Algorithm 2 unifies $\mathit{atom}(v)\delta_{j-1} = B_{i,j}\delta_{j-1}$ with a fresh variant $p(\overline{t}')\sigma'$ of $p(\overline{t}')$, where σ' is a renaming substitution, resulting in an mgu γ (by (4.23), $B_{i,j}\delta_{j-1}$ and $p(\overline{t}')\sigma'$ are unifiable) and then transfers the subquery $(\overline{s}'_{j-1}\gamma, (\delta_{j-1}\gamma)_{|\mathit{post\_vars}(v)})$ through $(v, \mathit{succ}(v))$. Let

$$\overline{s}'_j = \overline{s}'_{j-1}\gamma \;\text{ and }\; \delta_j = (\delta_{j-1}\gamma)_{|\mathit{post\_vars}(v)}. \tag{4.24}$$

We have that $\sigma = \sigma'\sigma''$ for some renaming substitution σ'' such that

$$\sigma'' \text{ does not use variables of } \overline{s}'_{j-1},\ \delta_{j-1} \text{ and } \mathit{pre\_vars}(v). \tag{4.25}$$

Thus $B_{i,j}\delta_{j-1}\sigma''\theta'_{k_j} = B_{i,j}\delta_{j-1}\theta'_{k_j}$, and by (4.23) and the fact $\sigma = \sigma'\sigma''$, we have that

$$(B_{i,j}\delta_{j-1})\sigma''\theta'_{k_j} = B_{i,j}\delta_{j-1}\theta'_{k_j} = p(\bar{t}')\sigma\theta'_{k_j} = (p(\bar{t}')\sigma')\sigma''\theta'_{k_j}.$$

Hence, $B_{i,j}\delta_{j-1}$ and $p(\bar{t}')\sigma'$ are unifiable using $\sigma''\theta'_{k_j}$, while γ is an mgu for them. Hence

$$\sigma''\theta'_{k_j} = \gamma\mu' \tag{4.26}$$

for some substitution μ'. Let $\eta' = \mu'\mu$. We have that:

$$\begin{array}{rl} & (\bar{s}', (B_{i,j+1}, \ldots, B_{i,n_i}))\varrho\theta_1 \ldots \theta_{k_{j+1}-1} \\ = & ((\bar{s}', (B_{i,j+1}, \ldots, B_{i,n_i}))\varrho\theta_1 \ldots \theta_{k_j-1})\theta_{k_j} \ldots \theta_{k_{j+1}-1} \\ = & (\bar{s}'_{j-1}, (B_{i,j+1}, \ldots, B_{i,n_i})\delta_{j-1})\eta\theta_{k_j} \ldots \theta_{k_{j+1}-1} \ \text{ (by the assumption (4.17))} \\ = & (\bar{s}'_{j-1}, (B_{i,j+1}, \ldots, B_{i,n_i})\delta_{j-1})\theta'_{k_j} \ldots \theta'_{k_{j+1}-1}\mu \ \text{ (by (4.21))} \\ = & (\bar{s}'_{j-1}, (B_{i,j+1}, \ldots, B_{i,n_i})\delta_{j-1})\sigma''\theta'_{k_j} \ldots \theta'_{k_{j+1}-1}\mu \ \text{ (by (4.25))} \\ = & (\bar{s}'_{j-1}, (B_{i,j+1}, \ldots, B_{i,n_i})\delta_{j-1})\gamma\mu'\mu \ \text{ (by (4.22) and (4.26))} \\ = & (\bar{s}'_{j}, (B_{i,j+1}, \ldots, B_{i,n_i})\delta_{j})\eta' \ \text{ (by (4.24) and the fact } \eta' = \mu'\mu). \end{array}$$

We have shown (4.18) and thus proved Remark 4.2 for the case when the predicate of $B_{i,j}$ is extensional.

Now consider the case when the predicate p of $B_{i,j}$ is an intensional predicate.

By the assumption (4.19), the subquery $(\bar{s}'_{j-1}, \delta_{j-1})$ was also added to $unprocessed\_subqueries_2(v)$. Let $B_{i,j}\delta_{j-1} = p(\bar{t}'_j)$. If $T(p) = true$ (resp. $T(p) = false$) then the pair $(\bar{t}'_j, \bar{t}'_j)$ (resp. tuple $\bar{t}'_j$) was transferred through the edge $(v, input\_p)$, hence there must exist some tuple pair $(\bar{t}, \bar{t}')$ (resp. tuple $\bar{t}'$) that was added to $tuple\_pairs(input\_p)$ (resp. $tuples(input\_p)$) at some step such that $(\bar{t}, \bar{t}')$ (resp. $\bar{t}'$) is more general than a fresh variant of $(\bar{t}'_j, \bar{t}'_j)$ (resp. $\bar{t}'_j$), and thus $(\bar{t}, \bar{t}')\lambda = (\bar{t}'_j, \bar{t}'_j)\lambda'$ (resp. $\bar{t}'\lambda = \bar{t}'_j\lambda'$) for some substitution λ that uses only variables from $\bar{t}, \bar{t}'$ (resp. $\bar{t}'$) and a renaming substitution λ' with domain contained in $Vars(\bar{t}'_j)$. Hence, $(\bar{t}, \bar{t}')\alpha = (\bar{t}'_j, \bar{t}'_j)$ (resp. $\bar{t}'\alpha = \bar{t}'_j$) for the substitution $\alpha = \lambda(\lambda')^{-1}$. We can assume that α uses only variables from $\bar{t}, \bar{t}'$ and $\bar{t}'_j$ (resp. $\bar{t}'$ and $\bar{t}'_j$). Thus,

$$B_{i,j}\delta_{j-1} = p(\bar{t}'_j) = p(\bar{t}')\alpha \quad \text{if } T(p) = false, \tag{4.27}$$

and

$$B_{i,j}\delta_{j-1} = p(\bar{t}'_j) = p(\bar{t})\alpha = p(\bar{t}')\alpha \quad \text{if } T(p) = true. \tag{4.28}$$

By (4.21) and Lifting Lemma 2.2, it follows that there exists a refutation of $P \cup I \cup \{\leftarrow p(\bar{t})\}$ if $T(p) = true$ (resp. $P \cup I \cup \{\leftarrow p(\bar{t}')\}$ if $T(p) = false$) using the leftmost selection function and mgu's $\theta''_{k_j}, \ldots, \theta''_{k_{j+1}-1}$ such that the term-depths of the goals are not greater than l and

$$\alpha\theta'_{k_j} \ldots \theta'_{k_{j+1}-1} = \theta''_{k_j} \ldots \theta''_{k_{j+1}-1}\beta \tag{4.29}$$

for some substitution β. By the inductive assumption, $tuples(ans\_p)$ contains a tuple $\bar{t}''$ such that $\bar{t}'\theta''_{k_j} \ldots \theta''_{k_{j+1}-1}$ is an instance of a variant of $\bar{t}''$. Since

$$\begin{array}{rcll} B_{i,j}\delta_{j-1}\theta'_{k_j} \ldots \theta'_{k_{j+1}-1} & = & p(\bar{t}')\alpha\theta'_{k_j} \ldots \theta'_{k_{j+1}-1} & \text{(by (4.27) and (4.28))} \\ & = & p(\bar{t}')\theta''_{k_j} \ldots \theta''_{k_{j+1}-1}\beta & \text{(by (4.29)),} \end{array}$$

it follows that

$$B_{i,j}\delta_{j-1}\theta'_{k_j}\dots\theta'_{k_{j+1}-1} \text{ is an instance of a variant of } p(\overline{t}''). \tag{4.30}$$

From a certain moment there were both $(\overline{s}'_{j-1}, \delta_{j-1}) \in \mathit{subqueries}(v)$ and $\overline{t}'' \in \mathit{tuples}(\mathit{ans\_p})$. Hence, at some step Algorithm 2 unified $\mathit{atom}(v)(\delta_{j-1}) = B_{i,j}\delta_{j-1}$ with a fresh variant $p(\overline{t}'')\sigma$ of $p(\overline{t}'')$, where σ is a renaming substitution. The atom $p(\overline{t}'')\sigma$ does not contain variables of $\overline{s}'_{j-1}$, δ_{j-1}, $\mathit{pre\_vars}(v)$ and $\theta'_{k_j}\dots\theta'_{k_{j+1}-1}$. By (4.30), $B_{i,j}\delta_{j-1}$ and $p(\overline{t}'')\sigma$ are unifiable. Let the resulting mgu be γ and let

$$\overline{s}'_j = \overline{s}'_{j-1}\gamma \text{ and } \delta_j = (\delta_{j-1}\gamma)_{|\mathit{post\_vars}(v)}. \tag{4.31}$$

Algorithm 2 then transferred the subquery $(\overline{s}'_j, \delta_j)$ through $(v, \mathit{succ}(v))$.

By (4.30), $B_{i,j}\delta_{j-1}\theta'_{k_j}\dots\theta'_{k_{j+1}-1}$ is an instance of $p(\overline{t}'')\sigma$. Let ρ be a substitution with domain contained in $\mathit{Vars}(p(\overline{t}'')\sigma)$ such that $B_{i,j}\delta_{j-1}\theta'_{k_j}\dots\theta'_{k_{j+1}-1} = p(\overline{t}'')\sigma\rho$. We have that

$$\begin{array}{l}\text{the domain of } \rho \text{ does not contain variables of } \overline{s}'_{j-1},\ \delta_{j-1},\ \mathit{pre\_vars}(v)\\ \text{and } \theta'_{k_j}\dots\theta'_{k_{j+1}-1}\end{array} \tag{4.32}$$

and $\theta'_{k_j}\dots\theta'_{k_{j+1}-1} \cup \rho$ is a unifier for $B_{i,j}\delta_{j-1}$ and $p(\overline{t}'')\sigma$. As γ is an mgu for $B_{i,j}\delta_{j-1}$ and $p(\overline{t}'')\sigma$, we have that

$$\gamma\mu' = (\theta'_{k_j}\dots\theta'_{k_{j+1}-1} \cup \rho) \tag{4.33}$$

for some substitution μ'. Let $\eta' = \mu'\mu$. We have that:

$$\begin{array}{rl} & (\overline{s}', (B_{i,j+1},\dots,B_{i,n_i}))\varrho\theta_1\dots\theta_{k_{j+1}-1}\\ = & (\overline{s}'_{j-1}, (B_{i,j+1},\dots,B_{i,n_i})\delta_{j-1})\theta'_{k_j}\dots\theta'_{k_{j+1}-1}\mu \ \ \text{(as shown before)}\\ = & (\overline{s}'_{j-1}, (B_{i,j+1},\dots,B_{i,n_i})\delta_{j-1})(\theta'_{k_j}\dots\theta'_{k_{j+1}-1}\cup\rho)\mu \ \ \text{(by (4.32))}\\ = & (\overline{s}'_{j-1}, (B_{i,j+1},\dots,B_{i,n_i})\delta_{j-1})\gamma\mu'\mu \ \ \text{(by (4.33))}\\ = & (\overline{s}'_j, (B_{i,j+1},\dots,B_{i,n_i})\delta_j)\eta' \ \ \text{(by (4.31) and the fact } \eta' = \mu'\mu). \end{array}$$

We have shown (4.18) and thus proved Remark 4.2 for the case when the predicate of $B_{i,j}$ is intensional. This completes the proof of this remark. ■

Consider the case when φ_i is a tail-recursive clause. Let $\theta_1,\dots,\theta_h$ be the mgu's used up to the step of deriving the goal $\leftarrow r(B'_{i,n_i}\theta_1\dots\theta_h)$. Thus, $k_{n_i} = h+1$.

By (4.14), after processing the atom $B'_{i,j-1}$ for $2 \leq j \leq n_i$, the next goal of the refutation of $\leftarrow r(\overline{s})$ is $\leftarrow (B'_{i,j},\dots,B'_{i,n_i})\theta_1\dots\theta_{k_j-1}$.

Recall that $\overline{s}'_0 = \overline{s}'\gamma_0$ and $\delta_0 = (\gamma_0)_{|\mathit{post\_vars}(\mathit{pre\_filter}_i)}$ and $k_1 = 2$. The subquery $(\overline{s}'_0, \delta_0)$ was transferred through the edge $(\mathit{pre\_filter}_i, \mathit{filter}_{i,1})$.

Consider the case $n_i > 1$. Observe that the premises of Remark 4.2 hold for $j = 1$ and for the subquery $(\overline{s}'_0, \delta_0)$ using $\eta = \eta_0$. Hence there exists a subquery $(\overline{s}'_1, \delta_1)$ that was transferred through $(\mathit{filter}_{i,1}, \mathit{succ}(\mathit{filter}_{i,1}))$ at some step and a substitution η_1 such that

$$(\overline{s}', (B_{i,2},\dots,B_{i,n_i}))\varrho\theta_1\dots\theta_{k_2-1} = (\overline{s}'_1, (B_{i,2},\dots,B_{i,n_i})\delta_1)\eta_1.$$

For each $1 < j < n_i$, we can apply Remark 4.2 to obtain a subquery $(\overline{s}'_j, \delta_j)$ and η_j (for η'). It follows that, when $j = n_i - 1$, the subquery $(\overline{s}'_{n_i-1}, \delta_{n_i-1})$ was transferred through the edge $(\mathit{filter}_{i,n_i-1}, \mathit{filter}_{i,n_i})$ and

$$(\overline{s}', B_{i,n_i})\varrho\theta_1 \dots \theta_{k_{n_i}-1} = (\overline{s}'_{n_i-1}, B_{i,n_i}\delta_{n_i-1})\eta_{n_i-1} \tag{4.34}$$

for some substitution η_{n_i-1}, which implies

$$\overline{s}'\varrho\theta_1 \dots \theta_{k_{n_i}-1} = \overline{s}'_{n_i-1}\eta_{n_i-1} \tag{4.35}$$

and

$$B_{i,n_i}\varrho\theta_1 \dots \theta_{k_{n_i}-1} = B_{i,n_i}\delta_{n_i-1}\eta_{n_i-1}. \tag{4.36}$$

Consider the case $n_i = 1$. The subquery $(\overline{s}'_0, \delta_0)$ was transferred through the edge $(\mathit{pre\_filter}_i, \mathit{filter}_{i,1})$. Since $\delta_0 = (\gamma_0)_{|\mathit{post\_vars}(\mathit{pre\_filter}_i)}$ and $\mathit{post\_vars}(\mathit{pre\_filter}_i) = \mathit{Vars}(B_{i,1})$ and by (4.15), it follows that $B_{i,1}\varrho\theta_1 = B_{i,1}\gamma_0\eta_0 = B_{i,1}\delta_0\eta_0$. Together with (4.16), it implies that (4.35) and (4.36) also hold for the case $n_i = 1$.

Let $B_{i,n_i}\delta_{n_i-1} = r(\overline{t}_{n_i})$. At some step, Algorithm 2 transfered the tuple pair $(\overline{t}_{n_i}, \overline{s}'_{n_i-1})$ through the edge $(\mathit{filter}_{i,n_i}, \mathit{input\_r})$, hence there must exist some tuple pair $(\overline{t}, \overline{t}')$ that was added to $\mathit{tuple\_pairs}(\mathit{input\_r})$ at some step such that $(\overline{t}, \overline{t}')$ is more general than a fresh variant of $(\overline{t}_{n_i}, \overline{s}'_{n_i-1})$, and thus $(\overline{t}, \overline{t}')\lambda = (\overline{t}_{n_i}, \overline{s}'_{n_i-1})\lambda'$ for some substitution λ that uses only variables from $\overline{t}$, $\overline{t}'$ and a renaming substitution λ' with domain contained in $\mathit{Vars}(\overline{t}_{n_i}) \cup \mathit{Vars}(\overline{s}'_{n_i-1})$. Hence, $(\overline{t}, \overline{t}')\alpha = (\overline{t}_{n_i}, \overline{s}'_{n_i-1})$ for the substitution $\alpha = \lambda(\lambda')^{-1}$. We can assume that α uses only variables from $\overline{t}$, $\overline{t}'$, $\overline{t}_{n_i}$ and $\overline{s}'_{n_i-1}$. Thus,

$$B_{i,n_i}\delta_{n_i-1} = r(\overline{t}_{n_i}) = r(\overline{t})\alpha, \tag{4.37}$$

and

$$\overline{s}'_{n_i-1} = \overline{t}'\alpha. \tag{4.38}$$

Take $\beta = \alpha\eta_{n_i-1}$. Since $B'_{i,n_i} = B_{i,n_i}\varrho$ and by (4.36), we have that:

$$\begin{array}{rcl} (\leftarrow B'_{i,n_i}\theta_1 \dots \theta_{k_{n_i}-1}) & = & (\leftarrow B_{i,n_i}\varrho\theta_1 \dots \theta_{k_{n_i}-1}) \\ & = & (\leftarrow B_{i,n_i}\delta_{n_i-1}\eta_{n_i-1}) \text{ (by (4.36))} \\ & = & (\leftarrow r(\overline{t})\alpha\eta_{n_i-1}) \text{ (by (4.37))} \\ & = & (\leftarrow r(\overline{t})\beta) \text{ (by the fact } \beta = \alpha\eta_{n_i-1}). \end{array} \tag{4.39}$$

Since the term-depth of $r(\overline{t})\beta = B'_{i,n_i}\theta_1 \dots \theta_{k_{n_i}-1}$ is not greater than l, the term-depth of $r(\overline{t})$ is also not greater than l. By (4.14), (4.39) and Lifting Lemma 2.2, there exists a refutation of $P \cup I \cup \{\leftarrow r(\overline{t})\}$ using the leftmost selection function and mgu's $\theta'_{k_{n_i}}, \dots, \theta'_{k_{n_i+1}-1}$ such that the term-depths of goals are not greater than l and

$$\beta\theta_{k_{n_i}} \dots \theta_{k_{n_i+1}-1} = \theta'_{k_{n_i}} \dots \theta'_{k_{n_i+1}-1}\mu \tag{4.40}$$

for some substitution μ. By the inductive assumption, $\mathit{tuples}(\mathit{ans\_r})$ contains a tuple $\overline{s}''$ such that

$$\overline{t}'\theta'_{k_{n_i}} \dots \theta'_{k_{n_i+1}-1} \text{is an instance of a variant of } \overline{s}''. \tag{4.41}$$

Recall that $k_{n_i+1} = y + 1$. Hence,

$$\begin{aligned}
\overline{s}'\theta_1 \ldots \theta_y &= \overline{s}'\theta_1 \ldots \theta_{k_{n_i-1}}\theta_{k_{n_i}} \ldots \theta_{k_{n_i+1}-1} \\
&= \overline{s}'\varrho\theta_1 \ldots \theta_{k_{n_i-1}}\theta_{k_{n_i}} \ldots \theta_{k_{n_i+1}-1} \;\; \text{(by (4.13))} \\
&= \overline{s}'_{n_i-1}\eta_{n_i-1}\theta_{k_{n_i}} \ldots \theta_{k_{n_i+1}-1} \;\; \text{(by (4.35))} \\
&= \overline{t}'\alpha\eta_{n_i-1}\theta_{k_{n_i}} \ldots \theta_{k_{n_i+1}-1} \;\; \text{(by (4.38))} \\
&= \overline{t}'\beta\theta_{k_{n_i}} \ldots \theta_{k_{n_i+1}-1} \;\; \text{(by the fact } \beta = \alpha\eta_{n_i-1}) \\
&= \overline{t}'\theta'_{k_{n_i}} \ldots \theta'_{k_{n_i+1}-1}\mu \;\; \text{(by (4.40))}.
\end{aligned}$$

This together with (4.41) implies that $\overline{s}'\theta_1 \ldots \theta_y$ is an instance of a variant of $\overline{s}''$. By (4.11), it follows that $\overline{s}'\theta$ is an instance of a variant of $\overline{s}''$.

We now consider the case when φ_i is not a tail-recursive clause. By (4.14), after processing the atom $B'_{i,j-1}$ for $2 \leq j \leq n_i + 1$, the next goal of the refutation of $\leftarrow r(\overline{s})$ is $\leftarrow (B'_{i,j}, \ldots, B'_{i,n_i})\theta_1 \ldots \theta_{k_j-1}$. (If $j = n_i + 1$ then the goal is empty.)

Similarly to the previous case, $(\overline{s}'_0, \delta_0)$ is a subquery Algorithm 2 transferred through $(\mathit{pre\_filter}_i, \mathit{filter}_{i,1})$ and observe that the premises of Remark 4.2 hold for $j = 1$ and for the subquery $(\overline{s}'_0, \delta_0)$ using $\eta = \eta_0$. Hence, there exist a subquery $(\overline{s}'_1, \delta_1)$ that was transferred through $(\mathit{filter}_{i,1}, \mathit{succ}(\mathit{filter}_{i,1}))$ at some step and a substitution η_1 such that

$$(\overline{s}', (B_{i,2}, \ldots, B_{i,n_i}))\varrho\theta_1 \ldots \theta_{k_2-1} = (\overline{s}'_1, (B_{i,2}, \ldots, B_{i,n_i})\delta_1)\eta_1.$$

For each $1 < j \leq n_i$, we can apply Remark 4.2 to obtain a subquery $(\overline{s}'_j, \delta_j)$ and η_j (for η'). Since $\mathit{post\_vars}(\mathit{filter}_{i,n_i}) = \emptyset$, it follows that, for $j = n_i$, $(\overline{s}'_{n_i}, \varepsilon)$ is a subquery that was transferred through $(\mathit{filter}_{i,n_i}, \mathit{post\_filter}_i)$ at some step and

$$\overline{s}'\varrho\theta_1 \ldots \theta_{k_{n_i+1}-1} = \overline{s}'_{n_i}\eta_{n_i}.$$

Since $k_{n_i+1} = y + 1$ and by (4.11) and (4.13), it follows that

$$\overline{s}'\theta = \overline{s}'\theta_1 \ldots \theta_y = \overline{s}'\varrho\theta_1 \ldots \theta_y = \overline{s}'_{n_i}\eta_{n_i}.$$

Thus, $\overline{s}'\theta$ is an instance of $\overline{s}'_{n_i}$. Since $(\overline{s}'_{n_i}, \varepsilon)$ was transferred through the edge $(\mathit{filter}_{i,n_i}, \mathit{post\_filter}_i)$, $\mathit{tuples}(\mathit{ans\_r})$ will contain $\overline{s}''$ such that $\overline{s}'_{n_i}$ is an instance of a fresh variant of $\overline{s}''$. It follows that, $\overline{s}'\theta$ is an instance of a variant of $\overline{s}''$. This completes the proof of this lemma. ∎

Theorem 4.4 (Completeness). *After a run of Algorithm 2 (using parameter l) on a query $(P, q(\overline{x}))$ and an extensional instance I, for every SLD-refutation of $P \cup I \cup \{\leftarrow q(\overline{x})\}$ that uses the leftmost selection function, does not contain any goal with term-depth greater than l and has a computed answer θ with term-depth not greater than l, there exists $\overline{s} \in \mathit{tuples}(\mathit{ans\_q})$ such that $\overline{x}\theta$ is an instance of a variant of $\overline{s}$.*

This theorem immediately follows from Lemma 4.3. Together with Theorem 2.1 (on completeness of SLD-resolution) it makes a relationship between correct answers of $P \cup I \cup \{\leftarrow q(\overline{x})\}$ and the answers computed by Algorithm 2 for the query $(P, q(\overline{x}))$ on the extensional instance I. ∎

For queries and extensional instances without function symbols, we take term-depth bound $l = 0$ and obtain the following completeness result, which immediately follows from the above theorem.

Corollary 4.5. *After a run of Algorithm 2 using $l = 0$ on a query $(P, q(\overline{x}))$ and an extensional instance I that do not contain function symbols, for every computed answer θ of an SLD-refutation of $P \cup I \cup \{\leftarrow q(\overline{x})\}$ that uses the leftmost selection function, there exists $\overline{t} \in tuples(ans\_q)$ such that $\overline{x}\theta$ is an instance of a variant of $\overline{t}$.* ∎

4.1.3 Data Complexity

In this section, the *data complexity* of Algorithm 2 is estimated, which is measured w.r.t. the size of the extensional instance I when the query $(P, q(\overline{x}))$ and the term-depth bound l are fixed. The estimation is very similar to the one given in [45] for QSQN. We include it here to make the text self-contained.

If terms are represented as sequences of symbols or as trees then there will be a problem with complexity. Namely, unifying the terms $f(x_1, \ldots, x_n)$ and $f(g(x_0, x_0), \ldots, g(x_{n-1}, x_{n-1}))$, we get a term of exponential length. Another example is the pair $f(x_1, \ldots, x_n, x_1, \ldots, x_n)$ and $f(y_1, \ldots, y_n, g(y_0, y_0), \ldots, g(y_{n-1}, y_{n-1}))$. If the term-depth bound l is used in all steps, including the ones of unification, then the problem will not arise. But we do not want to be so restrictive.

To represent a term we use instead a rooted acyclic directed graph which is permitted to have multiple ordered arcs and caches nodes representing the same subterm. Such a graph will simply be called a DAG. As an example, the DAG of $f(x, a, x)$ has the root n_f labeled by f, a node n_x labeled by x, a node n_a labeled by a, and three ordered edges outgoing from n_f: the first one and the third one are connected to n_x, while the second one is connected to n_a.

The *size of a term* t, denoted by $size(t)$, is defined to be the size of the DAG of t (i.e., the number of nodes and edges of the DAG of t). The sizes of other term-based expressions or data structures are defined as usual. For example, we define:

- the *size of a tuple* $(t_1, \ldots, t_k)$ to be $size(t_1) + \ldots + size(t_k)$,
- the *size of a set of tuples* to be the sum of the sizes of those tuples,
- the *size of a substitution* $\{x_1/t_1, \ldots, x_k/t_k\}$ to be $k + size(t_1) + \ldots + size(t_k)$,
- the *size of a node* v *of a QSQN-TRE* (V, E, T, C) to be the sum of the sizes of the components of $C(v)$.

Using DAGs to represent terms, unification of two atoms A and A' can be done in polynomial time in the sizes of A and A'. In the case A and A' are unifiable, the resulting atom and the resulting mgu have sizes that are polynomial in the sizes of A and A'. Similarly, checking whether A is an instance of A' can also be done in polynomial time in the sizes of A and A'.

The following theorem estimates the data complexity of Algorithm 2, under the assumption that terms are represented by DAGs and unification and checking instances of atoms are done in polynomial time.

Theorem 4.6. *For a fixed query and a fixed bound l on term-depth, Algorithm 2 runs in polynomial time in the size of the extensional instance.*

Proof. Consider a run of Algorithm 2 using parameter l on a query $(P, q(\overline{x}))$ and on an extensional instance I with size n. Here, $(P, q(\overline{x}))$ and l are fixed. Thus, for every

$1 \leq i \leq m$, n_i is bounded by a constant. Similarly, if p is an intensional predicate from P then the arity of p is also bounded by a constant.

Observe that the number of tuples (resp. tuple pairs) that are added to any set of the form $tuples(input\_p)$ (resp. $tuple\_pairs(input\_p)$) or $tuples(ans\_p)$ is bounded by a polynomial of n. The reasons are:

- intensional predicates come from P,
- constant symbols and function symbols come from P and I,
- $tuples(input\_p)$ (resp. $tuple\_pairs(input\_p)$) and $tuples(ans\_p)$ consist of tuples (resp. tuple pairs) with term-depth bounded by l,
- a tuple (resp. tuple pair) is added to a set of the form $tuples(input\_p)$ (resp. $tuple\_pairs(input\_p)$) or $tuples(ans\_p)$ only when its fresh variant is not an instance of any tuple from the set,
- a tuple (resp. tuple pair) is deleted from a set of the form $tuples(input\_p)$ (resp. $tuple\_pairs(input\_p)$) or $tuples(ans\_p)$ only when it is an instance of a new tuple added to the set.

For similar reasons, the number of subqueries that are added to any set of the form $subqueries(v)$ is also bounded by a polynomial of n.

Consequently, the sizes of sets of the form $tuples(input\_p)$, $tuple\_pairs(input\_p)$, $tuples(ans\_p)$, $subqueries(v)$, $unprocessed(v, w)$, $unprocessed\_subqueries(v)$, $unprocessed\_subqueries_2(v)$ or $unprocessed\_tuples(v)$ are bounded by a polynomial of n. Therefore, the size of the constructed QSQN-TRE is bounded by a polynomial of n, and any execution of procedure `transfer2`, procedure `fire2` or function `active-edge` is done in polynomial time in n.

A transfer or a firing for an edge (u, v) is done only when a new tuple (resp. tuple pair) was added to $tuples(u)$ (reps. $tuple\_pairs(u)$) or a new subquery was added to $subqueries(u)$. Thus, we can conclude that Algorithm 2 runs in polynomial time in n. ■

Corollary 4.7. *Algorithm 2 with term-depth bound $l = 0$ is a complete evaluation algorithm with polynomial time data complexity for the class of queries over a signature without function symbols.*

This corollary follows from Lemma 4.2 (on soundness), Corollary 4.5 (on completeness) and the above theorem (on data complexity). ■

4.2 QSQN with Right/Tail-Recursion Elimination

This section proposes an evaluation method called QSQN-rTRE for evaluating queries to Horn knowledge bases, which can eliminate not only tail-recursive predicates but also intensional predicates that appear rightmost in the bodies of the program clauses. Particularly, the rightmost intensional predicates in the bodies of the program clauses can be processed in the same way as for tail-recursive predicates discussed in Section 4.1. Thus, in this section, a program clause is said to be *right/tail-recursive* if it is either a tail-recursive clause or a clause with an intensional predicate that appears rightmost in the body of that program clause.

4.2.1 Definitions

Definition 4.4. We say that a predicate p *directly rightmost-depends* on a predicate q if p directly depends on q and q appears rightmost in the bodies of some program clauses defining p. We define the relation *rightmost-depends* to be the transitive closure of "directly rightmost-depends". ∎

Let P be a positive logic program and $\varphi_1, \ldots, \varphi_m$ be all the program clauses of P, with $\varphi_i = (A_i \leftarrow B_{i,1}, \ldots, B_{i,n_i})$, for $1 \leq i \leq m$ and $n_i \geq 0$. The following definition shows how to make a QSQ-Net structure with right/tail-recursion elimination from the given positive logic program P.

Definition 4.5 (QSQN-rTRE Structure). A *query-subquery net structure with right/tail-recursion elimination* (*QSQN-rTRE structure* for short) of P is a tuple (V, E, T) defined as in the case of QSQN-TRE (see Definition 4.2), except that:

- for $1 \leq i \leq m$, $post\_filter_i \in V$ iff either φ_i is not right/tail-recursive or $T_{idb}(p) = false$, where p is the predicate of A_i,
- E also contains $(post\_filter_i, ans\_p_k)$, for each $1 \leq i \leq m$, $1 \leq k \leq m$ and $k \neq i$ such that $post\_filter_i$ exists and the predicate p_k of A_k rightmost-depends on the predicate p_i of A_i. ∎

As for QSQN-TRE, $T(v)$ denotes $T_{edb}(v)$ if v is a node $filter_{i,j}$ such that $B_{i,j}$ is an extensional predicate, and $T(p)$ denotes $T_{idb}(p)$ for an intensional predicate p. Thus, T can be called a *memorizing type* for extensional predicates (as in QSQ-net structures), and a *right/tail-recursion-elimination type* for intensional predicates. We call the pair (V, E) the *QSQN-rTRE topological structure* of P with respect to T_{idb}.

Example 4.3. The upper part of Figure 4.5 illustrates a logic program and its QSQN-TRE topological structure w.r.t. T_{idb} with $T_{idb}(q) = true$ and $T_{idb}(p) = T_{idb}(s) = false$, where q, p, s are intensional predicates, t is an extensional predicate, x, y, z are variables and a is a constant symbol. The lower part depicts the QSQN-rTRE topological structure of the same program w.r.t. T_{idb} with $T_{idb}(q) = T_{idb}(p) = T_{idb}(s) = true$. ∎

Definition 4.6 (QSQN-rTRE). A *query-subquery net with right/tail-recursion elimination* (*QSQN-rTRE* for short) of P is a tuple $N = (V, E, T, C)$ such that (V, E, T) is a QSQN-rTRE structure of P, C is a mapping that associates each node $v \in V$ with a structure called the *contents* of v, which differs from the one for QSQN-TRE in the following:

- If $v = input\_p$ and $T(p) = true$ then $C(v)$ consists of:
  - $ta\_pairs(v)$: a set of *tuple-atom pairs* $(\bar{t}, q(\bar{t}'))$, where $\bar{t}$ is a generalized tuple of the same arity as p and $q(\bar{t}')$ is an atom. A tuple-atom pair $(\bar{t}, q(\bar{t}'))$ means that we are trying to solve the atom $p(\bar{t})$, and any found answer substitution should generate an answer for $q(\bar{t}')$,
  - $unprocessed(v, w)$ for each $(v, w) \in E$: a subset of $ta\_pairs(v)$.

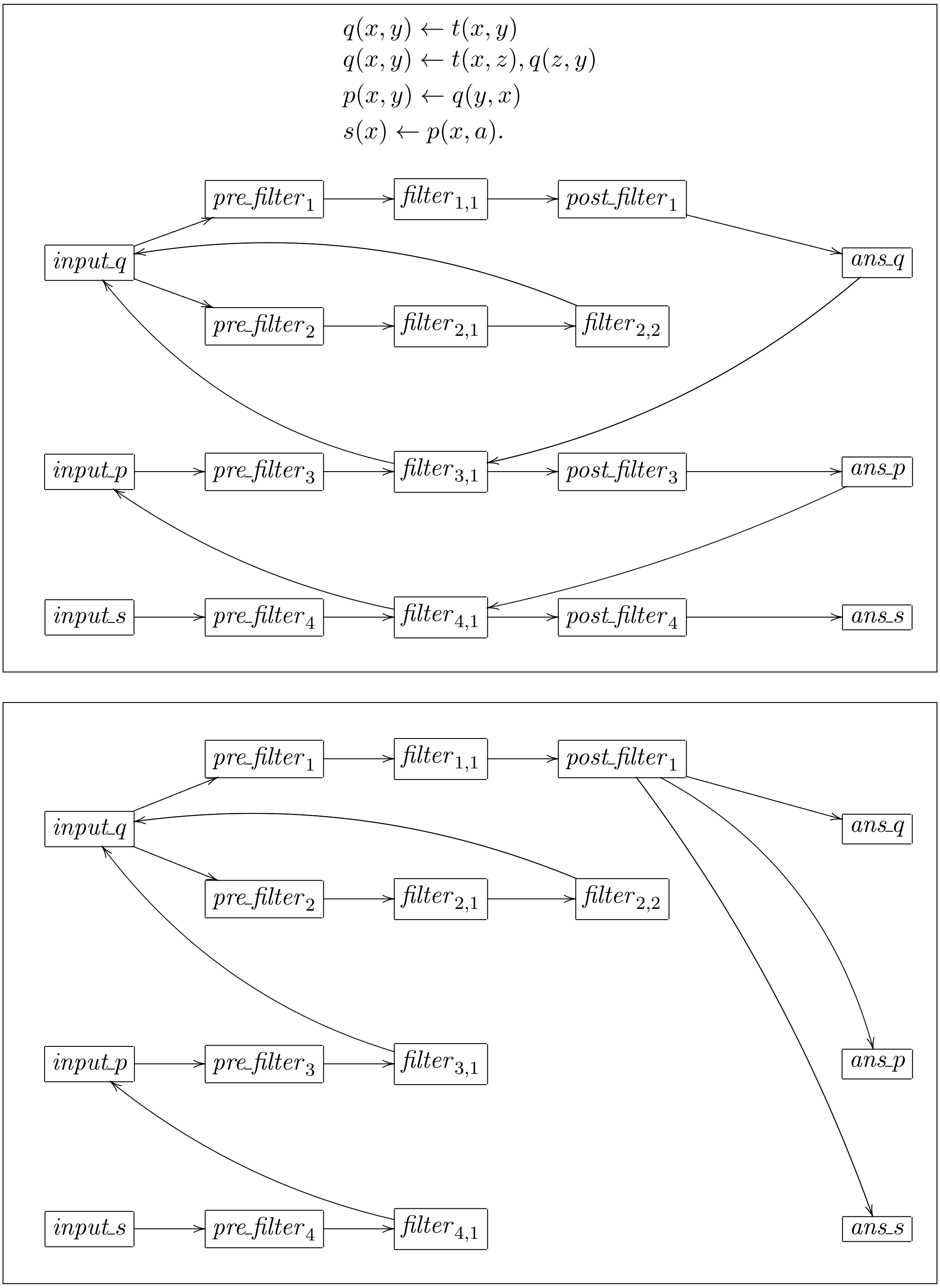


Fig. 4.5: The QSQN-TRE and QSQN-rTRE topological structures.

The notion of being empty is defined for QSQN-rTRE similarly as for QSQN-TRE, where the attribute $ta\_pairs(v)$ replaces the attribute $tuple\_pairs(v)$. ■

The notion of successor and the notations *succ* and $succ_2$ are defined similarly as for QSQN-TRE, except that: if v is $post\_filter_i$ then v may have more than one successor.

Now, a *subquery* is a pair of the form $(q(\bar{t}), \delta)$, where $q(\bar{t})$ is an atom and δ is an idempotent substitution such that $dom(\delta) \cap Vars(\bar{t}) = \emptyset$.

Remark 4.3. *For an intensional predicate p with $T(p) = true$, the intuition behind a tuple-atom pair $(\bar{t}, q(\bar{t}')) \in ta\_pairs(input\_p)$ is that:*

- *$\bar{t}$ is a usual input tuple for p, but the intended goal at a higher level is $\leftarrow q(\bar{t}')$,*
- *any correct answer for $P \cup I \cup \{\leftarrow p(\bar{t})\}$ is also a correct answer for $P \cup I \cup \{\leftarrow q(\bar{t}')\}$,*
- *if a substitution θ is a computed answer of $P \cup I \cup \{\leftarrow p(\bar{t})\}$ then we will store the tuple $\bar{t}'\theta$ in ans_q instead of storing the tuple $\bar{t}\theta$ in ans_p.* ■

We say that a pair $(\bar{t}, q(\bar{t}'))$ is *more general* than $(\bar{t}_2, q(\bar{t}_2'))$, and $(\bar{t}_2, q(\bar{t}_2'))$ is an *instance* of $(\bar{t}, q(\bar{t}'))$, if there exists a substitution θ such that $(\bar{t}, q(\bar{t}'))\theta = (\bar{t}_2, q(\bar{t}_2'))$.

For $v = filter_{i,j}$ and p being the predicate of A_i, the meaning of a subquery $(q(\bar{t}), \delta) \in subqueries(v)$ is as follows: if $T(p) = false$ (resp. $T(p) = true$) then there exists $\bar{s} \in tuples(input\_p)$ (resp. $(\bar{s}, q(\bar{s}')) \in ta\_pairs(input\_p)$) such that for processing the goal $\leftarrow p(\bar{s})$ using the program clause $\varphi_i = (A_i \leftarrow B_{i,1}, \ldots, B_{i,n_i})$, unification of $p(\bar{s})$ and A_i as well as processing of the subgoals $B_{i,1}, \ldots, B_{i,j-1}$ were done, amongst others, by using a sequence of mgu's $\gamma_0, \ldots, \gamma_{j-1}$ with the property that $\bar{t} = \bar{s}\gamma_0 \ldots \gamma_{j-1}$ and $q = p$ (resp. $\bar{t} = \bar{s}'\gamma_0 \ldots \gamma_{j-1}$) and $\delta = (\gamma_0 \ldots \gamma_{j-1})_{|Vars((B_{i,j}, \ldots, B_{i,n_i}))}$. Informally, a subquery $(q(\bar{t}), \delta)$ transferred through an edge to v is processed as follows:

- if $v = filter_{i,j}$, $kind(v) = extensional$ and $pred(v) = p$ then, for each $\bar{t}'' \in I(p)$, if $atom(v)\delta = B_{i,j}\delta$ is unifiable with a fresh variant of $p(\bar{t}'')$ by an mgu γ then transfer the subquery $(q(\bar{t})\gamma, (\delta\gamma)_{|post\_vars(v)})$ through $(v, succ(v))$,
- if $v = filter_{i,j}$, $kind(v) = intensional$, $pred(v) = p$ and either $(T(p) = false)$ or $(T(p) = true$ and $j < n_i)$ then
  - if $T(p) = false$ then transfer the tuple $\bar{t}'$ such that $p(\bar{t}') = atom(v)\delta = B_{i,j}\delta$ through $(v, input\_p)$ to add its fresh variant to $tuples(input\_p)$,
  - else if $j < n_i$ then transfer the pair $(\bar{t}', p(\bar{t}'))$ such that $p(\bar{t}') = atom(v)\delta = B_{i,j}\delta$ through $(v, input\_p)$ to add its fresh variant to $ta\_pairs(input\_p)$,
  - for each currently existing $\bar{t}' \in tuples(ans\_p)$, if $atom(v)\delta = B_{i,j}\delta$ is unifiable with a fresh variant of $p(\bar{t}')$ by an mgu γ then transfer the subquery $(q(\bar{t})\gamma, (\delta\gamma)_{|post\_vars(v)})$ through $(v, succ(v))$,
  - store the subquery $(q(\bar{t}), \delta)$ in $subqueries(v)$, and later, for each new $\bar{t}'$ added to $tuples(ans\_p)$, if $atom(v)\delta = B_{i,j}\delta$ is unifiable with a fresh variant of $p(\bar{t}')$ by an mgu γ then transfer the subquery $(q(\bar{t})\gamma, (\delta\gamma)_{|post\_vars(v)})$ through $(v, succ(v))$,
- if $v = filter_{i,n_i}$, $kind(v) = intensional$, $pred(v) = p$, $T(p) = true$ then transfer the pair $(\bar{t}', q(\bar{t}))$ such that $p(\bar{t}') = atom(v)\delta = B_{i,n_i}\delta$ through $(v, input\_p)$ to add its fresh variant to $ta\_pairs(input\_p)$,

Algorithm 3: for evaluating a query $(P, q(\overline{x}))$ on an extensional instance I.

let (V, E, T) be a QSQN-rTRE structure of P;
`// ` T ` can be chosen arbitrarily or appropriately`
set C so that $N = (V, E, T, C)$ is an empty QSQN-rTRE of P;
let $\overline{x}'$ be a fresh variant of $\overline{x}$;
if $T(q) = false$ **then**
  $tuples(input\_q) := \{\overline{x}'\}$;
  foreach $(input\_q, v) \in E$ **do** $unprocessed(input\_q, v) := \{\overline{x}'\}$;
else
  $ta\_pairs(input\_q) := \{(\overline{x}', q(\overline{x}'))\}$;
  foreach $(input\_q, v) \in E$ **do** $unprocessed(input\_q, v) := \{(\overline{x}', q(\overline{x}'))\}$;
while *there exists* $(u, v) \in E$ *such that* `active-edge`(u, v) *holds* **do**
  select $(u, v) \in E$ such that `active-edge`(u, v) holds;
  `// any strategy is acceptable for the above selection`
  `fire3`(u, v)
return $tuples(ans\_q)$

– if $v = post\_filter_i$ then transfer the subquery $(q(\overline{t}), \varepsilon)$ through $(post\_filter_i, ans\_q)$ to add $\overline{t}$ to $tuples(ans\_q)$.

Formally, like the case of QSQN and QSQN-TRE, the processing of a subquery or an input/answer tuple or an input pair in a QSQN-rTRE is designed so that:

– every subquery or input/answer tuple or input pair that is subsumed by another one or has a term-depth greater than a fixed bound l is ignored,

– the processing is divided into smaller steps which can be delayed at each node to maximize adjustability and allow various control strategies,

– the processing is done set-at-a-time (e.g., for all the unprocessed subqueries accumulated in a given node).

Algorithm 3 (on page 57) repeatedly selects an active edge and fires the operation for the edge. All of the related functions and procedures used for Algorithm 3 are presented in Appendix D. In particular, Algorithm 3 uses the function `active-edge`(u, v) (on page 28), which returns *true* if the data accumulated in u can be processed to produce some data to transfer through the edge (u, v). If `active-edge`(u, v) is *true* then the procedure `fire3`(u, v) (on page 116) processes the data accumulated in u that has not been processed before and transfers appropriate data through the edge (u, v). This procedure uses the procedures `add-tuple` (on page 27), `add-ta-pair` (on page 116), `add-subquery3`, `compute-gamma3` (on page 115) and `transfer3` (on pages 117-118). The procedure `transfer3`(D, u, v) specifies the effects of transferring data D through the edge (u, v) of a QSQN-rTRE.

Note: Regarding the QSQN-rTRE topological structure of the logic program given in Example 4.3 (illustrated in Figure 4.5), for the query $s(x)$, after producing a set of (answer) tuples, Algorithm 3 only adds these tuples to $tuples(ans\_s)$. Thus, no tuple is added to $tuples(ans\_p)$ and $tuples(ans\_q)$. In this case, we can exclude the nodes ans_p, ans_q and the related edges from the net, but we keep them for answering other queries of the form $p(\ldots)$ or $q(\ldots)$.

4.2.2 Properties of Algorithm 3

We present below properties of Algorithm 3. We omit their proofs as they are analogous to the ones we have given for Lemma 4.2 (on soundness), Theorem 4.4 (on completeness) and Theorem 4.6 (on data complexity) for QSQN-TRE.

Soundness: After a run of Algorithm 3 on a query $(P, q(\overline{x}))$ and an extensional instance I, for every intensional predicate p of P, every computed answer $\overline{t} \in tuples(ans\_p)$ is a correct answer in the sense that $P \cup I \models \forall(p(\overline{t}))$.

Completeness: After a run of Algorithm 3 (using parameter l) on a query $(P, q(\overline{x}))$ and an extensional instance I, for every SLD-refutation of $P \cup I \cup \{\leftarrow q(\overline{x})\}$ that uses the leftmost selection function, does not contain any goal with term-depth greater than l and has a computed answer θ with term-depth not greater than l, there exists $\overline{s} \in tuples(ans\_q)$ such that $\overline{x}\theta$ is an instance of a variant of $\overline{s}$.

Data Complexity: For a fixed query and a fixed bound l on term-depth, Algorithm 3 runs in polynomial time in the size of the extensional instance.

Chapter 5

Incorporating Stratified Negation into QSQN

Positive logic programs can express only monotonic queries. As many queries of practical interest are non-monotonic, it is desirable to consider normal logic programs, which allow negation to occur in the bodies of program clauses. Much research has been done on the semantics of normal logic programs, for instance, stratified semantics [2] (for stratified logic programs), stable-model semantics [30] and well-founded semantics [29]. In this chapter, we incorporate the concept of stratified negation into query-subquery nets to obtain an evaluation method called QSQN-STR for dealing with the class of stratified logic programs that are safe with respect to the leftmost selection function. Roughly speaking, a stratified logic program can be divided into a number of layers (called strata) such that they are evaluated sequentially. To have a firm "yes" for a negative literal $\sim B$ when processing the body of a program clause[1], we should already have all answers for B in a previous stage when processing an earlier stratum of the program. For evaluating queries to stratified logic programs, we use QSQN-STR together with control strategies that are admissible w.r.t. strata's stability. We also apply a term-depth bound for atoms, subqueries and substitutions occurring in derivations.

The rest of this chapter is organized as follows. We give definitions for stratified knowledge bases and the related ones in Section 5.1. The notion of QSQN-STR and our Algorithm 4 for evaluating queries to stratified knowledge bases are specified in Section 5.2. Some properties of Algorithm 4 are provided and proved in Section 5.3. Preliminary experiments for QSQN-STR are provided and discussed later in Chapter 6.

5.1 Notions and Definitions

In this section, we define the notions of safe logic programs, stratified logic programs, stratified knowledge bases and their semantics. We now give some definitions and recall the related notions of [37, 47].

Definition 5.1 (Safe Logic Program). A *safe program clause (w.r.t. the leftmost*

[1] Here, $\sim$ denotes negation w.r.t. a non-monotonic semantics.

selection function) is an expression of the form $A \leftarrow B_1, \ldots, B_k$ with $k \geq 0$, such that:

- A is an atom and each B_i is a literal, where negation is now denoted by $\sim$ instead of $\neg$ (to emphasize the non-monotonic semantics),
- every variable occurring in A occurs also in $B_1, \ldots, B_k$,
- every variable occurring in a negative literal B_j in the body of a program clause occurs also in some positive literals B_i in the body of that clause such that $i < j$.

A *safe logic program (w.r.t. the leftmost selection function)* is a finite set of safe program clauses. ∎

From now on in this chapter, by a "clause" we mean a safe program clause.

Definition 5.2 (Semi-positive Logic Program). A safe logic program is called a *semi-positive logic program* if it allows negation to appear in the bodies of program clauses only before atoms of extensional predicates. ∎

Definition 5.3 (Stratification). Given a safe logic program P, a *stratification* of P is a partition $P = P_1 \cup \ldots \cup P_n$ such that for each $1 \leq i \leq n$, we have the following properties:

- if an intensional predicate p occurs in a positive literal of a clause from P_i, then the clauses defining p must belong to $P_1 \cup \ldots \cup P_i$,
- if an intensional predicate p occurs in a negative literal of a clause from P_i with $i > 1$, then the clauses defining p must belong to $P_1 \cup \ldots \cup P_{i-1}$.

Each P_i is called a *stratum* of the stratification. ∎

Definition 5.4 (Stratified Logic Program). A safe logic program is called a *stratified logic program* if it has a stratification. ∎

Note that by the definition a program can admit several stratifications. In this chapter, by a "program" we mean a stratified logic program (which may be a semi-positive program) that is safe w.r.t. the leftmost selection function.

Definition 5.5 (Stratified Knowledge Base). A *stratified knowledge base* is defined to be a pair (P, I), where P is a stratified logic program for defining intensional predicates and I is an instance of extensional predicates. ∎

5.2 QSQN with Stratified Negation

Let P be a stratified logic program and $\varphi_1, \ldots, \varphi_m$ be all the program clauses of P, with $\varphi_i = (A_i \leftarrow B_{i,1}, \ldots, B_{i,n_i})$, for $1 \leq i \leq m$ and $n_i \geq 0$.

Definition 5.6 (QSQN-STR Structure). A *query-subquery net structure with stratified negation* of P, also called a *QSQN-STR structure* of P, is a tuple (V, E, T) defined as in the case of QSQN (see Definition 3.1) with the following modification:

- for each intensional predicate p and each $1 \leq i \leq m$ and $1 \leq j \leq n_i$ such that $B_{i,j}$ is an atom of p, the pair $(ans\_p, filter_{i,j})$ is an edge (i.e., belongs to E) iff $B_{i,j}$ is a positive literal.

$$
\begin{aligned}
&path(x, y) \leftarrow edge(x, y)\\
&path(x, y) \leftarrow edge(x, z), path(z, y)\\
&acyclic(x, y) \leftarrow path(x, y), \sim path(y, x).
\end{aligned}
$$

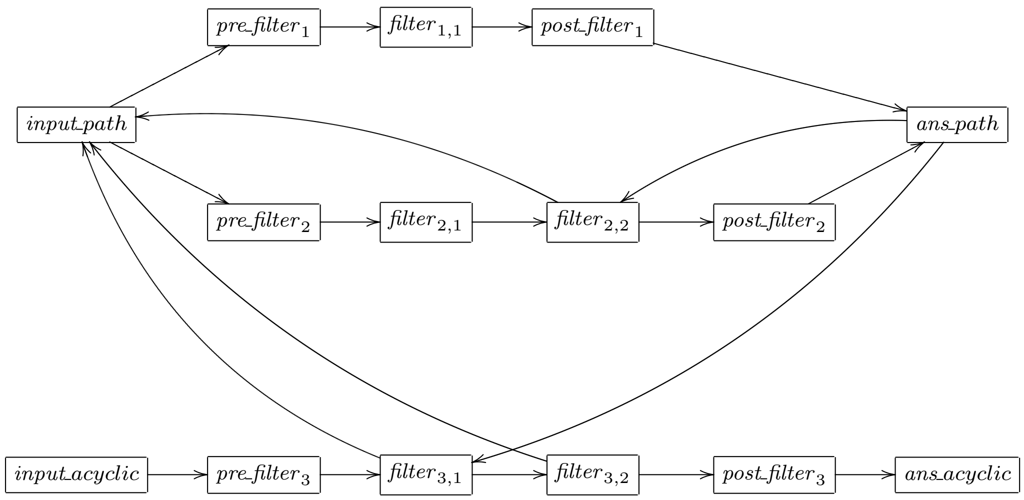


Fig. 5.1: The QSQN-STR topological structure of the program given in Example 5.1.

The pair (V, E) is called the *QSQN-STR topological structure* of P. ∎

Example 5.1. This example is a stratified logic program, which defines whether a node is connected to another in a directed graph, but not vice versa. It is taken from [51]. Figure 5.1 illustrates the program and its QSQN-STR topological structure, where *path* and *acyclic* are intensional predicates, *edge* is an extensional predicate, x, y and z are variables. ∎

Definition 5.7 (QSQN-STR). A *query-subquery net with stratified negation* of P, also called a *QSQN-STR* of P, is a tuple $N = (V, E, T, C)$ such that (V, E, T) is a QSQN-STR structure of P, and C is a mapping that associates each node $v \in V$ with a structure called the *contents* of v, which differs from the one for QSQN in the following:

– If $v = \mathit{filter}_{i,j}$ and p is the predicate of $B_{i,j}$ then
  - $C(v)$ also contains $neg(v)$, where $neg(v) = \mathit{true}$ if $B_{i,j}$ is a negative literal, and $neg(v) = \mathit{false}$ otherwise,
  - $atom(v)$ is redefined as follows: $atom(v) = B_{i,j}$ if $B_{i,j}$ is a positive literal, and $atom(v) = B'$ if $B_{i,j} = \sim B'$,
  - in the case p is intensional and $neg(v) = \mathit{true}$: $\mathit{unprocessed\_tuples}(v)$ is empty and can thus be ignored.

The notion of being empty is defined for QSQN-STR similarly as for QSQN. ∎

Definition 5.8. Given a stratified logic program $P = P_1 \cup \ldots \cup P_n$ and a QSQN-STR (V, E, T, C) of P, we say that a node $v \in V$ "*belongs to*" the layer k if v is constructed by some program clauses in P_k.[2] In that case, we say that the layer number of v is k, denoted by $layer(v) = k$. ∎

Definition 5.9 (Stability of a Layer). A QSQN-STR is said to be *stable* up to a layer k if every edge (u, v) of that structure such that the layer numbers of u and v are less than or equal to k is not active, where the activeness is defined by the function `active-edge4` (on page 119), which is similar to the one for QSQN. ∎

Definition 5.10 (Admissibility w.r.t. Strata's Stability). A control strategy for a given QSQN-STR is said to be *admissible w.r.t. strata's stability* if before firing any edge $(v, succ(v))$ such that $v = filter_{i,j}$, $layer(v) = k$, $pred(v) = p$ and p is an intensional predicate with $layer(input\_p) = h < k$, the QSQN-STR is stable up to the layer h and the edges $(v, input\_p)$ and (ans_p, v) are not active. ∎

Algorithm 4 (on page 63) evaluates a query to a stratified knowledge base. It repeatedly selects an active edge and fires the operation for the edge. All the the related functions and procedures used for Algorithm 4 are presented in Appendix E. In particular, Algorithm 4 uses the function `active-edge4`(u, v) (on page 119), which returns *true* if the data accumulated in u can be processed to produce some data to transfer through the edge (u, v). If `active-edge4`(u, v) is *true* then the procedure `fire4`(u, v) (on page 120) processes the data accumulated in u that has not been processed before and transfers appropriate data through the edge (u, v). This procedure uses the procedure `transfer4`(D, u, v) (on page 121), which specifies the effects of transferring data D through the edge (u, v) of a QSQN-STR.

The following remark states a property of Algorithm 4. It follows from the safety condition of P. The proof is straightforward and omitted.

Remark 5.1. *For every intensional predicate r used in P, if $\bar{t} \in tuples(ans\_r)$ then $\bar{t}$ is a ground tuple (i.e., a tuple without variables).* ∎

Recall that a subquery is a pair of the form $(\bar{t}, \delta)$, where $\bar{t}$ is a generalized tuple and δ is an idempotent substitution such that $dom(\delta) \cap Vars(\bar{t}) = \emptyset$. Informally, Algorithm 4 differs from Algorithm 1 (for QSQN) in "firing" an edge (u, v) as follows:

- If $v = filter_{i,j}$, $neg(v) = true$, $kind(v) = extensional$ and $T(v) = false$ then for every subquery $(\bar{t}, \delta)$ transferred through the edge (u, v), if $atom(v)\delta \notin \{p(\bar{t}') \mid \bar{t}' \in I(p)\}$[3] then transfer the subquery $(\bar{t}, \delta_{|post\_vars(v)})$ through the edge $(v, succ(v))$, which is shown in Steps 32 and 33 of the procedure `transfer4` (on page 121).
- If $u = filter_{i,j}$, $neg(u) = true$, $v = succ(u)$, and either $kind(u) = intensional$ or both $kind(u) = extensional$ and $T(u) = true$ then for every subquery $(\bar{t}, \delta)$ from $subqueries(filter_{i,j})$, if $atom(u)\delta \notin \{p(\bar{t}') \mid (\bar{t}' \in I(p)$ if $kind(u) = extensional)$ or $(\bar{t}' \in tuples(ans\_p)$ if $kind(u) = intensional)\}$[4] then transfer the subquery $(\bar{t}, \delta_{|post\_vars(u)})$ through the edge (u, v), which is shown in Steps 12, 13, 28 and 29 of the function `fire4` (on page 120).

[2] I.e., P_k contains a clause φ_i such that v is of the form $input\_p$, pre_filter_i, $filter_{i,j}$, $post\_filter_i$, or ans_p, where p is the predicate of A_i.

[3] In this case, $atom(v)\delta$ is a ground atom.

[4] In this case, $atom(v)\delta$ is a ground atom and $tuples(ans\_p)$ contains only ground tuples.

Algorithm 4: for evaluating a query $(P, q(\overline{x}))$ on an extensional instance I for the stratified logic program P that is safe w.r.t. the leftmost selection function.

let (V, E, T) be a QSQN-STR structure of P;
`//` T `can be chosen arbitrarily or appropriately`
set C so that $N = (V, E, T, C)$ is an empty QSQN-STR of P;

let $\overline{x}'$ be a fresh variant of $\overline{x}$;
tuples(*input_q*) $:= \{\overline{x}'\}$;
foreach $(input\_q, v) \in E$ **do** *unprocessed*(*input_q*, v) $:= \{\overline{x}'\}$;

while *there exists* $(u, v) \in E$ *such that* `active-edge4`(u, v) *holds* **do**
  select any edge $(u, v) \in E$ such that `active-edge4`(u, v) holds and the selection satisfies the admissibility w.r.t. strata's stability;
  `fire4`(u, v);

return *tuples*(*ans_q*)

Example 5.2. The aim of this example is to illustrate how Algorithm 4 works step by step. The program P in this example is a modified version of Example 1.1, which is specified as follows, where p, q_1 and q_2 are intensional predicates, r_1, r_2 and s are extensional predicates, x, y and z are variables:

$$
\begin{aligned}
&q_1(x, y) \leftarrow r_1(x, y)\\
&q_1(x, y) \leftarrow r_1(x, z), q_1(z, y)\\
&q_2(x, y) \leftarrow r_2(x, y)\\
&q_2(x, y) \leftarrow r_2(x, z), q_2(z, y)\\
&p(x, y) \leftarrow s(x, y), \sim q_1(x, y), \sim q_2(x, y).
\end{aligned}
$$

The query is $p(x, y)$ and the extensional instance I is specified as follows, where a_i and $b_{i,j}$ are constant symbols and $m = n = 30$:

$$
\begin{array}{rcl}
I(r_1) & = & \{(a_i, a_{i+1}) \mid 0 \leq i < m\},\\
I(r_2) & = & \{(a_0, b_{1,j}) \mid 1 \leq j \leq n\} \cup\\
 & & \{(b_{i,j}, b_{i+1,j}) \mid 1 \leq i < m-1 \textit{ and } 1 \leq j \leq n\} \cup\\
 & & \{(b_{m-1,j}, a_m) \mid 1 \leq j \leq n\},\\
I(s) & = & \{(a_0, a_m), (a_0, a_{m+1})\}.
\end{array}
$$

We give below a trace of running Algorithm 4 for Example 5.2. Figure 5.2 illustrates the QSQN-STR topological structure of the program P. For convenience, we name the edges of the net by E_i ($1 \leq i \leq 30$) as shown in this figure. Assume that Algorithm 4 evaluates the query $p(x, y)$ to the program P and the extensional instance I using a control strategy that selects active edges for "firing" as follows.

1. Algorithm 4 starts with an empty QSQN-STR and then adds a fresh variant (x_1, y_1) of (x, y) to the empty sets *tuples*(*input_p*) and *unprocessed*(E_1). This makes the edge E_1 to become active.

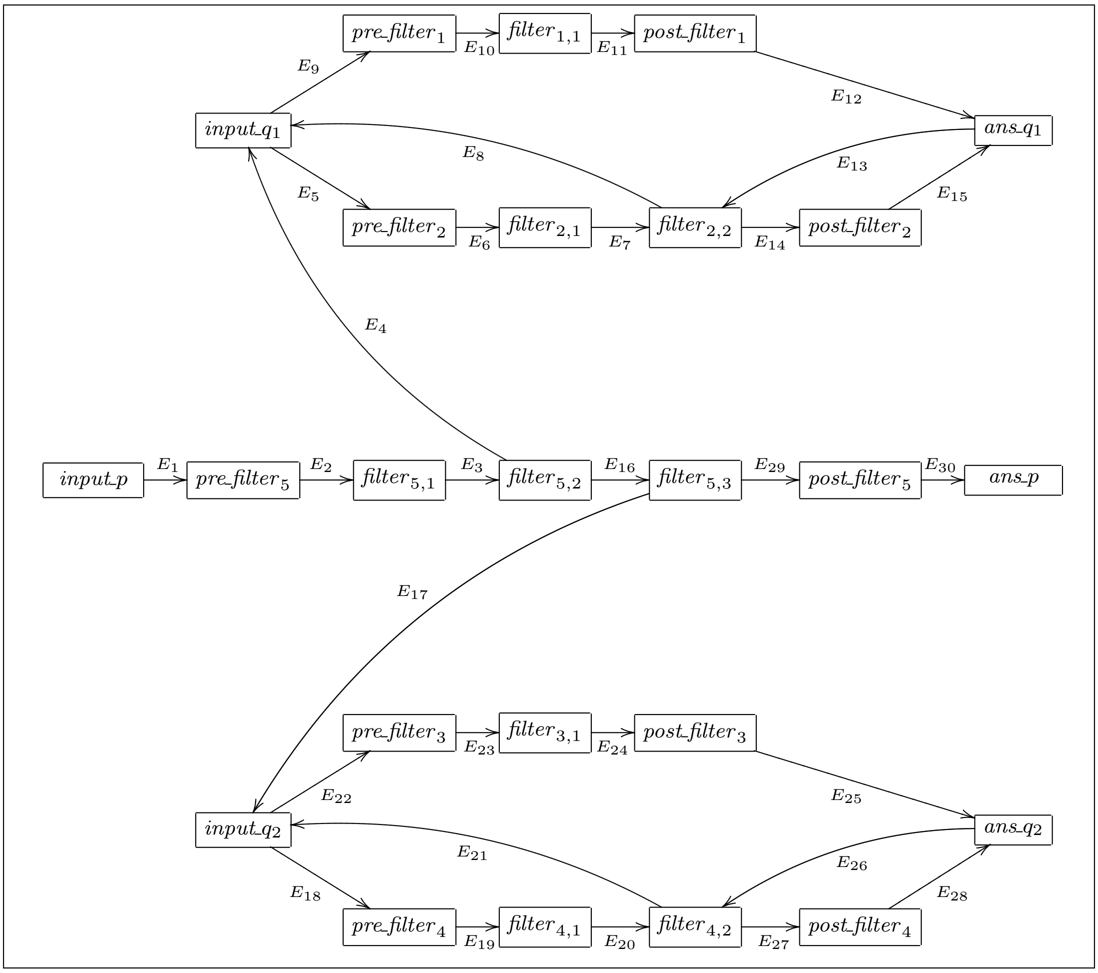


Fig. 5.2: The QSQN-STR topological structure of the program given in Example 5.2.

2. After processing *unprocessed*(E_1), the algorithm empties this set and transfers (x_1, y_1) through the edge E_1. This produces $\{((x_1, y_1), \{x/x_1, y/y_1\})\}$, which is then transferred through the edge E_2, producing $\{((a_0, a_{30}), \{x/a_0, y/a_{30}\}), ((a_0, a_{31}), \{x/a_0, y/a_{31}\})\}$, which in turn is then transferred through the edge E_3 and added to the empty sets *subqueries*$(\mathit{filter}_{5,2})$, *unprocessed_subqueries*$(\mathit{filter}_{5,2})$ and *unprocessed_subqueries*$_2(\mathit{filter}_{5,2})$. The edges E_4 and E_{16} become active.
3. After firing the active edge E_4, the algorithm adds the set of tuples $\{(a_0, a_{30}), (a_0, a_{31})\}$ to the empty sets *tuples*$(\mathit{input\_q}_1)$, *unprocessed*(E_5) and *unprocessed*(E_9). The edge E_4 is now inactive and the edges E_5 and E_9 become active.
4. After processing *unprocessed*(E_5), the algorithm empties this set and transfers $\{(a_0, a_{30}), (a_0, a_{31})\}$ through the edge E_5. This produces $\{((a_0, a_{30}), \{x/a_0, y/a_{30}\}), ((a_0, a_{31}), \{x/a_0, y/a_{31}\})\}$, which is then transferred through the edge E_6, producing $\{((a_0, a_{30}), \{y/a_{30}, z/a_1\}), ((a_0, a_{31}), \{y/a_{31}, z/a_1\})\}$, which in turn is then transferred through the edge E_7 and added to the empty sets *subqueries*$(\mathit{filter}_{2,2})$,

unprocessed_subqueries(*filter*$_{2,2}$) and *unprocessed_subqueries$_2$*(*filter*$_{2,2}$). The edge E_5 is now inactive and the edges E_8 and E_{14} become active.

5. After firing the active edge E_8, the algorithm adds the set of tuples $\{(a_1, a_{30}), (a_1, a_{31})\}$ to the sets *tuples*(*input_q*$_1$), *unprocessed*(E_5) and *unprocessed*(E_9).
6. The algorithm repeatedly fires the active edges E_5 and E_8 until no new tuple is added to *tuples*(*input_q*$_1$). After these steps, *tuples*(*input_q*$_1$) contains the set of tuples $\{(a_i, a_{30}), (a_i, a_{31}) \mid 0 \leq i \leq 30\}$. Next, the algorithm fires the active edge E_9 and adds the tuple (a_{29}, a_{30}) to the empty sets *tuples*(*ans_q*$_1$) and *unprocessed*(E_{13}). The edge E_9 is now inactive and the edge E_{13} becomes active. Then, the algorithm repeatedly fires the active edges E_{13} and E_{14} until no new tuple is added to *tuples*(*ans_q*$_1$). As the result of these steps, *tuples*(*ans_q*$_1$) contains the set of tuples $\{(a_i, a_{30}) \mid 0 \leq i < 30\}$.
7. The remaining active edge is E_{16}, where *unprocessed_subqueries*(*filter*$_{5,2}$) $=$ $\{((a_0, a_{30}), \{x/a_0, y/a_{30}\}),\ ((a_0, a_{31}), \{x/a_0, y/a_{31}\})\}$. After firing this active edge, since *tuples*(*ans_q*$_1$) contains the tuple (a_0, a_{30}), the algorithm only adds the subquery $\{((a_0, a_{31}), \{x/a_0, y/a_{31}\})\}$ to the empty sets *subqueries*(*filter*$_{5,3}$), *unprocessed_subqueries*(*filter*$_{5,3}$) and *unprocessed_subqueries$_2$*(*filter*$_{5,3}$). The edge E_{16} is now inactive and the edges E_{17} and E_{29} become active.
8. After firing the active edge E_{17}, the algorithm adds the tuple (a_0, a_{31}) to *tuples*(*input_q*$_2$), *unprocessed*(E_{18}) and *unprocessed*(E_{22}). The edge E_{17} is now inactive and the edges E_{18} and E_{22} become active.
9. The algorithm repeatedly fires the edges E_{18} and E_{21} until no new tuple is added to *tuples*(*input_q*$_2$). This makes the edge E_{27} to become active. After these steps, *tuples*(*input_q*$_2$) contains the set of tuples $\{(a_0, a_{31}), (a_{30}, a_{31})\} \cup \{(b_{i,j}, a_{31}) \mid 1 \leq i < 30 \text{ and } 1 \leq j \leq 30\}$. Next, the algorithm fires the active edge E_{22} without adding any tuple to *tuples*(*ans_q*$_2$). Since no tuple was added to *tuples*(*ans_q*$_2$), firing the edge E_{27} does not create data to be transferred. At this point, *tuples*(*ans_q*$_2$) $= \emptyset$.
10. The remaining active edge is E_{29}. Since no tuple was added to *tuples*(*ans_q*$_2$), after processing the set *unprocessed_subqueries*(*filter*$_{5,3}$), the algorithm makes the edge E_{29} to become inactive and transfers the subquery $\{((a_0, a_{31}), \varepsilon)\}$ through the edge E_{29}. This produces $\{(a_0, a_{31})\}$, which is then transferred through the edge E_{30} and added to the empty set *tuples*(*ans_p*).

At this point, no edge in the net is active. The algorithm terminates and returns the set *tuples*(*ans_p*) $= \{(a_0, a_{31})\}$. ∎

5.3 Soundness and Completeness of QSQN-STR for the Case without Function Symbols

In this section, we present the soundness and completeness of QSQN-STR for the case without function symbols. The case with function symbols is complicated and left for future work. We first introduce some definitions, which are based on [2, 37, 47].

Let (P, I) be a stratified knowledge base, (V, E, T) a QSQN-STR structure of P, and $P_1 \cup \ldots \cup P_n$ a stratification of P.

Definition 5.11 (Herbrand Base).

- The *Herbrand universe* for (P, I), denoted by $U_{P,I}$, is the set of all ground terms which are formed by using constants and function symbols in $P \cup I$;
- We define the *Herbrand base* for (P, I), denoted by $B_{P,I}$, to be the set of all ground atoms of the form $p(t_1, \ldots, t_n)$, where p is a predicate used in $P \cup I$ and each t_i belongs to $U_{P,I}$. ■

Definition 5.12 (Herbrand Interpretation). A *Herbrand interpretation* for (P, I) is a subset of the Herbrand base $B_{P,I}$. ■

Definition 5.13. Let $\mathcal{I}$ be a Herbrand interpretation. If $p(\bar{t})$ is a ground atom then:

$$\begin{array}{lcl} \mathcal{I}(p(\bar{t})) & \stackrel{def}{\equiv} & p(\bar{t}) \in \mathcal{I}, \\ \mathcal{I}(\sim p(\bar{t})) & \stackrel{def}{\equiv} & p(\bar{t}) \notin \mathcal{I}. \end{array}$$

■

Definition 5.14 (Immediate Consequence Operator). Let $ground(P \cup I)$ be the set of all ground instances of clauses in $P \cup I$ and $\mathcal{I}$ a Herbrand interpretation for (P, I). The *immediate consequence operator* of (P, I), denoted by $T_{P,I}$, is defined on $\mathcal{I}$ as follows:

$$T_{P,I}(\mathcal{I}) = \{A \mid A \leftarrow B_1, \ldots, B_k \in ground(P \cup I) \text{ and } \mathcal{I}(B_i) \text{ holds for all } 1 \leq i \leq k\}.$$

Let $T_{P,I} \uparrow \omega$ be defined as follows:

$$\begin{array}{rcl} T_{P,I} \uparrow 0 & = & I \\ T_{P,I} \uparrow (n+1) & = & T_{P,I}(T_{P,I} \uparrow n) \cup T_{P,I} \uparrow n, \quad \text{for } n \in \mathbb{N} \\ T_{P,I} \uparrow \omega & = & \bigcup_{n=0}^{\infty} T_{P,I} \uparrow n. \end{array}$$

■

Definition 5.15 (Standard Herbrand Model). Let $P_1 \cup \ldots \cup P_n$ be a stratification of P. We assume that $P_0 = \emptyset$. Let us set

$$\begin{array}{lcl} M_{\emptyset,I} & = & I \\ M_{P_1,I} & = & T_{P_1,I} \uparrow \omega \\ M_{P_1 \cup P_2,I} & = & T_{P_2,M_{P_1,I}} \uparrow \omega \\ & \vdots & \\ M_{P_1 \cup \ldots \cup P_n,I} & = & T_{P_n,M_{P_1 \cup \ldots \cup P_{n-1},I}} \uparrow \omega. \end{array}$$

We call $M_{P,I} = M_{P_1 \cup \ldots \cup P_n,I}$ the *standard Herbrand model* of $P \cup I$. ■

It is well-known that the standard Herbrand model $M_{P,I}$ does not depend on the chosen stratification for P (see, e.g., [2, Theorem 11]).

Lemma 5.1. *Let (P, I) be a stratified knowledge base without function symbols and let $P = P_1 \cup \ldots \cup P_n$ be a stratification of P. During a run of Algorithm 4 using $l = 0$, for every intensional predicate r of P with $layer(r) = k$ and for every tuple $\bar{t}$ and $\bar{t}'$,*

a) if $\bar{t} \in tuples(ans\_r)$ then $r(\bar{t}) \in M_{P,I}$,

b) if the QSQN-STR is stable up to layer k, $\bar{t} \in tuples(input\_r)$, $r(\bar{t}') \in M_{P,I}$ and $\bar{t}'$ is an instance of $\bar{t}$ then $\bar{t}' \in tuples(ans\_r)$.

Proof. For simplicity of the proof, $M_{P_1 \cup \ldots \cup P_k, I}$ will be denoted by M_k. We assume that $P_0 = \emptyset$. We prove this lemma by induction on the number k. The base case $k = 0$ is trivial. For the induction step, we first prove that the layer P_k can be treated as a positive logic program when considering p and $\sim p$, for p being any extensional predicate or intensional predicate defined in a lower layer, as extensional predicates specified by the interpretation M_{k-1}. For this, due to Remark 5.1 and the admissibility of the control strategy w.r.t. strata's stability, it is sufficient to show that, if $pred(A_i) = r$ and $pred(filter_{i,j}) = p$ with p defined in a layer with number h such that $h < k$ then:

(i) if $\bar{t} \in tuples(ans\_p)$ then $p(\bar{t}) \in M_{k-1}$,

(ii) if $(\bar{t}_{j-1}, \delta_{j-1}) \in subqueries(filter_{i,j})$ then every $p(\bar{t}) \in M_{k-1}$ such that $p(\bar{t})$ is an instance of $atom(filter_{i,j})\delta_{j-1}$ was added by Algorithm 4 to $tuples(ans\_p)$ at some step before the subquery $(\bar{t}_{j-1}, \delta_{j-1})$ is processed for the edge $(filter_{i,j}, succ(filter_{i,j}))$.

The assertion (i) follows from the inductive assumption (a) (note that $p(\bar{t}) \in M_{P,I}$ iff $p(\bar{t}) \in M_{k-1}$). For the assertion (ii), assume that $(\bar{t}_{j-1}, \delta_{j-1}) \in subqueries(filter_{i,j})$, $p(\bar{t}) \in M_{k-1}$ and $p(\bar{t})$ is an instance of $atom(filter_{i,j})\delta_{j-1}$. Before the edge $(filter_{i,j}, succ(filter_{i,j}))$ is "fired", the subquery $(\bar{t}_{j-1}, \delta_{j-1})$ has been processed for the edge $(filter_{i,j}, succ_2(filter_{i,j}))$ and, as a consequence, $tuples(input\_p)$ contains a tuple $\bar{t}''$ that is more general than a fresh variant of $atom(filter_{i,j})\delta_{j-1}$, and hence also more general than $p(\bar{t})$. At that moment, the QSQN-STR is stable up to the layer h. By the inductive assumption (b) for h instead of k and p, $\bar{t}''$, $\bar{t}$ instead of r, $\bar{t}$, $\bar{t}'$, respectively, we have that $\bar{t} \in tuples(ans\_p)$, which completes the proof of the assertion (ii).

Consider the assertion (a) and assume that the premise of the implication holds. Since $\bar{t} \in tuples(ans\_r)$, by Lemma 4.2 (on soundness of QSQN-TRE with $T(p) = false$ for every intensional predicate p), $\bar{t}$ is a correct answer for $P_k \cup M_{k-1} \cup \{\leftarrow r(\bar{t})\}$, treating P_k as a positive logic program in the way described above. Since SLD-resolution (the procedural semantics) "coincides" with the fixpoint semantics for positive logic program (see, e.g., [31, Theorem 7]), it follows that $r(\bar{t}) \in T_{P_k, M_{k-1}} \uparrow \omega$ and hence $r(\bar{t}) \in M_k$. This completes the proof of the assertion (a).

Consider the assertion (b) and assume that the premises of the implication hold. Since $r(\bar{t}') \in M_{P,I}$, we have that $r(\bar{t}') \in M_k$, which means $r(\bar{t}') \in T_{P_k, M_{k-1}} \uparrow \omega$. Since the fixpoint semantics for positive logic program "coincides" with the procedural semantics (SLD-resolution), there exits an SLD-refutation for $P_k \cup M_{k-1} \cup \{\leftarrow r(\bar{t}')\}$ with ε as the computed answer (treating P_k as a positive logic program).

Since $\bar{t}'$ is an instance of $\bar{t}$, there exists a substitution θ such that $\bar{t}' = \bar{t}\theta$. By Lifting Lemma 2.2, there exists an SLD-refutation for $P_k \cup M_{k-1} \cup \{\leftarrow r(\bar{t})\}$ with a computed answer θ' such that $\theta = \theta\varepsilon = \theta'\delta$ for some substitution δ. By Theorem 4.4 (on completeness of QSQN-TRE with $T(p) = \mathit{false}$ for every intensional predicate p), $\bar{t}\theta'$ is an instance of a fresh variant of some tuple $\bar{t}'' \in \mathit{tuples}(\mathit{ans\_r})$. Since $\bar{t}' = \bar{t}\theta = \bar{t}\theta'\delta$ is an instance of $\bar{t}\theta'$, $\bar{t}'$ is also an instance of $\bar{t}''$. Since $\bar{t}''$ is a ground tuple (by Remark 5.1), it follows that $\bar{t}' = \bar{t}''$. This completes the proof of the assertion (b). ∎

The following theorem immediately follows from the above lemma.

Theorem 5.2 (Soundness and Completeness). *Let (P, I) be a stratified knowledge base without function symbols. After a run of Algorithm 4 using $l = 0$ on a query $(P, q(\overline{x}))$ and the extensional instance I, for every tuple $\bar{t}$, $\bar{t} \in \mathit{tuples}(\mathit{ans\_q})$ iff $q(\bar{t}) \in M_{P,I}$.* ∎

Chapter 6

Preliminary Experiments

In this chapter, we present the experimental results and a discussion on the performance of the proposed methods. For this, we provide the IDFS control strategy, which is used for QSQN, QSQN-TRE and QSQN-rTRE, and another control strategy for QSQN-STR. Our experiments consider different kinds of logic programs, including non-recursive, tail recursive, non-tail recursive as well as logic programs with or without function symbols. We use typical examples from well-known articles related to deductive databases. We also provide new examples.

All of the experiments have been performed using our Java code [13] and extensional relations stored in a MySQL database. These experiments were performed on Windows Server 2008 (64 bit) with Intel(R) Xeon(R) CPU E5630 2×2.53 GHz and 8GB RAM. The package [13] also contains all of the reported experimental results. Our implementation allows queries of the form $q(\bar{t})$, where $\bar{t}$ is a tuple of terms.

This chapter is organized as follows. Section 6.1 proposes a control strategy, called IDFS. Sections 6.2, 6.3, 6.4 and 6.5 present the experimental settings and results for the QSQN, QSQN-TRE, QSQN-rTRE and QSQN-STR methods, respectively. In addition, we discuss the usefulness of the mentioned methods at the end of each section.

6.1 Improved Depth-First Control Strategy

Recall that in Algorithms 1, 2 and 3, we repeatedly select an active edge and fire the operation for it. Such a selection is decided by the adopted control strategy, which can be arbitrary. In [11, 45], we proposed the following control strategies:

- Disk Access Reduction (DAR), which tries to reduce the number of accesses to the secondary storage;
- Depth-First Search (DFS), which gives priority to the order of clauses in the positive logic program defining intensional predicates and thus allows the user to control the evaluation to a certain extent.

All of the experimental results in this dissertation were obtained without using the DAR and DFS control strategies. Thus, we omit the description of these strategies and refer the reader to [11, 45] for details.

In this section, we propose another control strategy called Improved Depth-First Control Strategy (IDFS), which is an improved version of DFS. The idea of the improvement is to enter deeper cycles in the considered net first and keep looping along the current "local" cycle as long as possible. This allows to accumulate as many as possible tuples or subqueries at a node before processing it.

Definition 6.1 (Priority). The *priority* of an edge (v, w) in a net (QSQN, QSQN-TRE or QSQN-rTRE) is a vector defined as follows:

- if $v = \mathit{input\_p}$ and $w = \mathit{pre\_filter}_i$ then $\mathit{priority}(v, w) = (a, b, c)$, where:
  - a is the truth value of (φ_i contains at least one intensional predicate r in the body),
  - if $a = \mathit{false}$ then $b = \mathit{false}$,
    else b is the truth value of (one of those predicates r depends on p)[1],
  - if $b = \mathit{false}$ then $c = 0$, else c is the modification timestamp of w,
- if $v = \mathit{ans\_p}$ and $w = \mathit{filter}_{i,j}$ then $\mathit{priority}(v, w) = (a, a', b, b', c)$, where:
  - a is the truth value of (p is the predicate of A_i),
    if ($a = \mathit{true}$) then a' is the truth value of (j is the smallest index such that $\mathit{pred}(\mathit{filter}_{i,j}) = p$), else $a' = \mathit{false}$,
  - b is the truth value of (p depends on the predicate of A_i)[2],
    if ($b = \mathit{true}$) then b' is the truth value of (j is the smallest index such that $\mathit{pred}(\mathit{filter}_{i,j}) = p$), else $b' = \mathit{false}$,
  - c is the modification timestamp of w,
- if v is $\mathit{filter}_{i,j}$ with $\mathit{kind}(v) = \mathit{intensional}$ then $\mathit{priority}(v, w) = (a)$, where $a = 2$ if $w = \mathit{succ}_2(v)$, and $a = 1$ otherwise,
- otherwise, $\mathit{priority}(v, w) = (1)$.

The priorities of two edges (v, w) and (v, w') are compared using the lexicographical order, where $\mathit{false} < \mathit{true}$. ■

Our IDFS control strategy follows the depth-first approach, but adopts a slight modification. It uses a stack of edges of the considered net (QSQN, QSQN-TRE or QSQN-rTRE) structure of P. Each of Algorithms 1, 2 and 3 together with this control strategy for evaluating a query $q(\overline{x})$ to a Horn knowledge base (P, I) runs as follows:

1. initialize $\mathit{input\_q}$ and the relations of the form $\mathit{unprocessed}(\mathit{input\_q}, v)$ appropriately, i.e.,
   - if the considered net is either QSQN or (QSQN-TRE with $T(q) = \mathit{false}$) or (QSQN-rTRE with $T(q) = \mathit{false}$) then
     - let $\overline{x}'$ be a fresh variant of $\overline{x}$ and set $\mathit{tuples}(\mathit{input\_q}) := \{\overline{x}'\}$,
     - for each edge $(\mathit{input\_q}, v)$ of the net do $\mathit{unprocessed}(\mathit{input\_q}, v) := \{\overline{x}'\}$,
   - else if the considered net is QSQN-TRE with $T(q) = \mathit{true}$ then
     - let $\overline{x}'$ be a fresh variant of $\overline{x}$ and set $\mathit{tuple\_pairs}(\mathit{input\_q}) := \{(\overline{x}', \overline{x}')\}$,

[1] Note that if $b = \mathit{true}$ then p and r mutually depend on each other.
[2] Note that if $a = \mathit{true}$ then p and the predicate of A_i mutually depend on each other.

- for each edge $(input\_q, v)$ of the net do $unprocessed(input\_q, v) := \{(\overline{x}', \overline{x}')\}$,

– else if the considered net is QSQN-rTRE with $T(q) = true$ then
  - let $\overline{x}'$ be a fresh variant of $\overline{x}$ and set $ta\_pairs(input\_q) := \{(\overline{x}', q(\overline{x}'))\}$,
  - for each edge $(input\_q, v)$ of the net do $unprocessed(input\_q, v) := \{(\overline{x}', q(\overline{x}'))\}$,

2. initialize the stack to the empty one and push all the edges outcoming from $input\_q$ into the stack in the increasing order w.r.t. their priorities (the lower the priority is, the earlier the edge is pushed into the stack),
3. while the stack is not empty do:
   (a) pop an edge (u, v) from the stack,
   (b) if (u, v) is an "active" edge then
   (b.1) if $u = ans\_p$, $v = filter_{i,j}$, $pred(v) = p$, p is not the predicate of A_i and there exist active edges (u', v') with $u' = input\_p$ then
     – push (u, v) into the stack,
     – set (u, v) to be the edge with the highest priority from those active edges (u', v'),
   (b.2) "fire" the edge (u, v),
   (b.3) push all the "active" edges outcoming from v into the stack in the increasing order w.r.t. their priorities,
   (b.4) if $v = filter_{i,j}$, $pred(v) = p$, the predicate of A_i is p, the edge $(v, succ_2(v))$[3] is not "active" and there exist active edges (u', v') with $u' = input\_p$ then
     – push (u', v'') into the stack, where (u', v'') is the edge with the highest priority from those active edges (u', v'),
4. return $tuples(ans\_q)$.

6.2 The QSQN method

In this section, we compare the QSQN, Magic-Sets and QSQR evaluation methods with respect to:

– the number of read/write operations on relations,
– the maximum number of tuples/subqueries kept in the computer memory,
– the number of accesses to the secondary storage when the memory is limited.

Other comparison results on the execution time are platform-dependent, and on the number of tuples/subqueries read from or written to the secondary storage are less representative. Thus, they are provided only online in [13].

6.2.1 Experimental Settings

For the Magic-Sets method, we implemented the Generalized Supplementary Magic Sets algorithm [1, 8]. In our implementation, the program obtained from the magic-sets transformation is evaluated by the improved semi-naive method [1]. The transformation is done using adornments and the method is sound and complete for Datalog queries (most of examples used in our experiments are Datalog queries). For application to Horn

[3] Recall that: if $pred(v) = p$ then $succ_2(v) = input\_p$.

knowledge bases, we also use a term-depth bound as in the case of QSQN and QSQR. The implemented Magic-Sets method using the term-depth bound is still sound[4], has the termination property, and returns the same set of results in answer relations as in the case of QSQN and QSQR for the performed tests.

Regarding the QSQR method, we implemented two algorithms, which use either the tuple-at-a-time technique or the set-at-a-time technique [39]. As the latter is more efficient, it is used for our comparison. For an appropriate comparison with the QSQN method, we modified the step 5 of the function `resolve-using-body-atom` on page 17 of [39] by checking the term-depth of a body atom before processing it.

We implemented the QSQN method together with the mentioned control strategies to obtain the corresponding variants QSQN-DAR, QSQN-DFS and QSQN-IDFS. The implemented QSQN-DAR method is not more efficient than the implemented QSQN-IDFS method. So, we only show the experimental results of QSQN-IDFS. For each test in our experiments, we set $T(v) = false$ for each $v = \mathit{filter}_{i,j} \in V$ with $\mathit{pred}(v) = \mathit{extensional}$ (so that subqueries for v are always processed immediately).

When processing a 0-ary predicate p (e.g., Example 1.1), as the answer is always binary (true or $false$), we break the computation for p as soon as we get the answer true when using any of the considered evaluation methods. This optimization technique can be generalized for other cases, but we leave it for future work.

We carry out experiments by two stages:

1. In the first stage, we assume that the computer memory is large enough to hold all the related extensional relations as well as the intermediate relations. During the query processing, for each operation of reading from a relation (resp. writing a set of tuples to a relation), we increase the counter of read (resp. write) operations on the relation by one. In a single task like firing an edge in QSQN or executing a rule in QSQR or Magic-Sets, if more than one read operations on a specific relation occur, we increase the counter of read operations on the relation only by one. For counting the maximum number of tuples/subqueries kept in the computer memory, we increase (resp. decrease) the counter of kept tuples/subqueries by one if a tuple/subquery is added to (resp. removed from) a relation. The returned value is the maximum value of this counter.
2. The second stage follows the first one. We limit the maximal number of tuples/subqueries that can be kept in the computer memory. When the limit is at low percentage, this usually requires load/unload operations from/to the secondary storage. The aim of this stage is to compute the number of accesses to the secondary storage when the computer memory is restricted. We test each example by using different limits, which are described in detail below. We also assume that the limited available memory is enough to store at least the biggest relation during the processing.

During the processing, whenever there is an action on a relation in the memory (e.g., loading/reading a relation, adding/removing a set of tuples or subqueries to/from a relation), we update its last access timestamp. If there is not enough available space for adding a set of tuples/subqueries to an in-memory relation, we have to unload an

[4] We did not study whether it is complete or not.

in-memory relation to the secondary storage. The question is: which relation among the ones in the memory should be unloaded first? In our experiments, the strategies for selecting a relation for unloading are based on the following criteria: *Timestamp*, *Relation-size* and *Extensional*. When used alone, a criterion has the following meaning:

1. *Timestamp*: unload a relation that has not been used in the longest period of time;
2. *Relation-size*: unload a relation with the biggest size;
3. *Extensional*: unload an extensional relation.

Our implementation of the methods uses a priority queue to specify which in-memory relation is selected for unloading. The user of our package [13] can choose a list of some among the above criteria. If more than one criteria are chosen, the preference reflects their order. For example, if the strategy is specified by [*Relation-size, Timestamp*], then the biggest in-memory relation is selected; if there are more than one in-memory relations with the biggest size, the one with the smallest last-access timestamp is selected for unloading. Of course, the relation used in the current single task has the lowest priority for unloading, regardless of the chosen strategy.

For counting the number of read/write operations on a relation, a predicate $magic\_p^{\alpha}$ (resp. p^{α}) in the Magic-Sets method is treated as the predicate *input_p* (resp. *ans_p*) in the QSQR and QSQN methods.[5] The number of accesses to *supplement* relations is defined to be the number of accesses to *subqueries* relations in QSQN, *sup_* relations in QSQR and *supmagic* relations in Magic-Sets. The other relations are treated as temporary relations. In the QSQN method, a relation of the form *unprocessed*, *unprocessed_subqueries*, $unprocessed\_subqueries_2$ or *unprocessed_tuples* can be implemented by a mark in the corresponding full relation (of the form *tuples* or *subqueries*). This saves memory and the status of whether the relation is empty or not can be kept by a flag. Thus, an execution of the function `active-edge` does not affect the number of accesses to the relations.

If a non-empty relation is loaded from the secondary storage to the computer memory, we increase the counter of read operations on the relation by one. Besides, if a relation that has been modified is unloaded from the computer memory, we save the relation to the secondary storage and increase the number of write operations on the relation by one. This means that unloading an extensional relation or a non-modified relation does not affect the counter.

The limit on the number of tuples/subqueries that can be kept in the computer memory is set as follows for each test. Let $m = max\{m_1, m_2, m_3\}$, where m_1, m_2 and m_3 are the maximum numbers of tuples/subqueries kept in the memory for QSQN, QSQR and Magic-Sets, respectively, in the case it is not restricted. Based on the value of m, we limit the maximal number of tuples/subqueries that can be kept in the memory sequentially to n_1, n_2 and n_3, where: $n_1 \approx 50\%m$, $n_2 \approx 30\%m$, and $n_3 \approx 20\%m$. In some cases, when all the mentioned methods cannot be run at $n_2 \approx 30\%m$ (using any of the strategies for unloading relations that are listed together with the test results), we use the following restrictions: $n_1 \approx 60\%m$, $n_2 \approx 45\%m$ and $n_3 \approx 30\%m$.

[5] An *adornment* for an m-ary predicate p is a string α of length m made up of b (bound) and f (free). By p^{α} we denote the predicate p adorned by α.

For the comparison between the QSQN, Magic-Sets and QSQR methods, we consider the following experiments:

Experiment 1. In this experiment, the mentioned methods are tested on datasets with different sizes (ranging from 400 to 10100 records (or tuples)), and their performances are compared in order to estimate how the experimental measures are affected by the size of the used datasets. Besides, the aim of Tests 6.1 and 6.2 is to show that, when the positive logic program defining intensional predicates is specified using the Prolog programming style, the QSQN-IDFS and QSQR methods (which use depth-first search) are usually more efficient than the Magic-Sets method (which uses the breadth-first search).

Test 6.1. This test uses the logic program, the extensional instance I and the query given in Example 1.1. The logic program is specified as follows, where p, q_1, q_2 are intensional predicates, r_1, r_2 are extensional predicates, x, y, z are variables and a_0, a_m are constant symbols:

$$\begin{array}{l} p \leftarrow q_1(a_0, a_m) \\ p \leftarrow q_2(a_0, a_m) \\ \\ q_1(x, y) \leftarrow r_1(x, y) \\ q_1(x, y) \leftarrow r_1(x, z), q_1(z, y) \\ \\ q_2(x, y) \leftarrow r_2(x, y) \\ q_2(x, y) \leftarrow r_2(x, z), q_2(z, y). \end{array}$$

The values of m and n for defining the extensional instance I are specified as follows:

(a) $m = n = 50$ (thus, r_1 has 50 tuples and r_2 has 2500 tuples),

(b) $m = n = 100$ (thus, r_1 has 100 tuples and r_2 has 10000 tuples).

As mentioned earlier, since the answer can only be either $true$ or $false$, we break the computation (for p) as soon as we get the answer $true$ when using any of the considered evaluation methods. After processing the tuple (a_0, a_m) in $input\_q_1$ and inserting an answer (the 0-ary tuple) into ans_q_1, the QSQN-IDFS method gets a positive answer for the query p and terminates the computation. The QSQR method uses iterative deepening search and terminates the computation after getting the answer $true$ for the query p from processing q_1. Now, consider the evaluation using the Magic-Sets method. After performing the magic-sets transformation and applying the generalized supplementary magic sets algorithm, the improved semi-naive evaluation method constructs a list $[R_1], \ldots, [R_n]$ of equivalence classes of intensional predicates w.r.t. their dependency [1]. Then, it computes the instances (relations) of the predicates in $[R_i]$ for each $1 \leq i \leq n$ in the increasing order, treating all the predicates in $[R_j]$ with $j < i$ as extensional predicates. The predicate p belongs to the last equivalence class $[R_n]$ and is only processed after finishing the computation for all the other predicates. That is why the QSQN-IDFS method (as well as the QSQR method) outperforms the Magic-Sets method on this test.

Test 6.2. This test uses the logic program that extends the logic program in Test 6.1 with the following clause for defining an intensional predicate s:

$$s(x, y) \leftarrow p, q_1(x, z), q_2(z, y).$$

It uses the same extensional instance I as in Test 6.1 and the query $s(x, y)$. Due to a similar reason as in Test 6.1, the QSQN-IDFS and QSQR methods outperform the Magic-Sets method on this test.

Test 6.2 may seem a bit artificial. However, the point is that breadth-first search is inflexible, and we believe that there are many meaningful queries for which depth-first evaluation beat breadth-first evaluation.

Regarding optimization techniques that allow to terminate evaluation of a subquery earlier, in the current implementation [13] we consider only the case when the subquery is an atom of a 0-ary predicate. Such optimization techniques can also be developed and implemented for the following cases:

1. the subquery is an atom without variables (i.e., a ground atom),
2. the subquery is the main query and the user just wants one or some answers,
3. the subquery is $p(\bar{t})$ and a tuple that is more general than $\bar{t}$ was inserted into ans_p.

Note the usefulness of the first case: when extending the QSQN method for dealing with logic programs with safe and stratified negation, at the time of processing a negative subquery, the subquery is without variables.

Test 6.3. This test uses the logic program from Example 3.1, which involves tail recursion as follows:

$$\begin{array}{l} p(x, y) \leftarrow q(x, y) \\ p(x, y) \leftarrow q(x, z), p(z, y). \end{array}$$

The extensional instance I for q is specified as follows:

$$\begin{array}{rcl} I(q) & = & \{(a_0, b_{1,j}) \mid 1 \leq j \leq n\} \cup \\ & & \{(b_{i,j}, b_{i+1,j}) \mid 1 \leq i < m \text{ and } 1 \leq j \leq n\}. \end{array}$$

We perform this test using the following queries:

$$(i) \;\; p(a_0, x), \qquad\qquad (ii) \;\; p(x, y).$$

Together with each mentioned query, we consider the following values of m and n:

$$(a) \;\; m = 5, n = 80; \qquad\qquad (b) \;\; m = 10, n = 150.$$

Experiment 2. This experiment includes the tests that concern the case with function symbols. For a function symbol f, by f^k we denote the nesting of f by k times. For example, $f^3(a) = f(f(f(a)))$. As mentioned earlier, we use a term-depth bound l for atoms and substitutions occurring in the computation to deal with function symbols. We consider the following tests for this experiment.

Test 6.4. This test is taken from [14] and specified as follows, where *path* is an intensional predicate, *edge* is an extensional predicate, x, y, z, w are variables, *nil* is a constant symbol standing for the empty list, and *cons* is a function symbol standing for the list constructor:

$$\begin{array}{l} path(x, y, cons(x, cons(y, nil))) \leftarrow edge(x, y) \\ path(x, y, cons(x, z)) \leftarrow edge(x, w), path(w, y, z). \end{array}$$

An atom $path(x, y, z)$ stands for "z is a list representing a path from x to y". We use the following extensional instance I for $edge$, where a - m are constant symbols:

$$\begin{array}{rcl} I(edge) & = & \{(a,b),(a,h),(b,c),(b,f),(c,d),(d,e),(e,c),(f,g), \\ & & (g,f),(h,i),(i,i),(j,c),(j,k),(k,d),(k,l),(l,m),(m,j)\}. \end{array}$$

The query is $path(x, d, y)$. We perform this test using the following term-depth bound l:

$$(a) \quad l = 20, \qquad\qquad (b) \quad l = 50.$$

Test 6.5. This test was given in [44]. It uses the following logic program:

$$\begin{array}{l} n(x,y) \leftarrow r(x,y) \\ n(x,y) \leftarrow q(f(w),y), n(z,w), p(x,f(z)) \\ s(x) \leftarrow n(c,x) \end{array}$$

where n, s are intensional predicates, p, q, r are extensional predicates, x, y, z, w are variables, f is a function symbol and c is a constant symbol. We use the term-depth bound $l = 10$ for this test (which can be changed before testing the package [13]). The query is $s(x)$ and the extensional instance is an extension of the one used in [44] and specified as follows, using $n = 20$:

$$\begin{array}{rcl} I(r) & = & \{(d,e)\} \\ I(p) & = & \{(c,f(d))\} \cup \{(b_i,f(c)),(c,f(b_i)) \mid 0 \le i \le n\} \\ I(q) & = & \{(f(e),a_0)\} \cup \{(f(a_i),b_i) \mid 0 \le i \le n\} \cup \{(f(b_i),a_{i+1}) \mid 0 \le i < n\} \cup \\ & & \{(f^k(b_i),f^k(a_{i+1})) \mid n \le i \le 2n-1,\ k = 2(i-n)+1\} \cup \\ & & \{(f^k(a_i),f^k(b_i)) \mid n+1 \le i \le 2n,\ k = 2(i-n)\}. \end{array}$$

Test 6.6. This test uses the following "*Same Generation*" logic program:

$$\begin{array}{l} sg(x,y) \leftarrow sibling(x,y) \\ sg(x,y) \leftarrow parent(x,z), sg(z,w), parent(y,w) \\ sibling(x,y) \leftarrow child(x,z), child(y,z) \\ parent(father(x),x) \\ parent(mother(x),x) \\ parent(x,y) \leftarrow child(y,x) \end{array}$$

where sg, $parent$ and $sibling$ are intensional predicates, $child$ is an extensional predicate, x, y, z, w are variables, $father$ and $mother$ are function symbols. The test uses the term-depth bound $l = 3$, the query $sg(x, y)$ and the extensional instance I for $child$ specified by $I(child) = \{(ann, john), (peter, john), (bill, john)\}$.

6.2.2 Results and Discussion

Table 6.1 (on page 77) presents a comparison between the QSQN (using the IDFS control strategy), Magic-Sets and QSQR evaluation methods w.r.t. the number of accesses to the extensional and intermediate relations as well as the maximum number of tuples/subqueries kept in the computer memory for Experiments 1 and 2. Tables 6.2

Tests		Methods	Reading (times) $inp\_/ans\_/sup\_/edb$	Writing (times) $inp\_/ans\_/sup\_$	Max No. of kept tuples
Test 6.1	(a)	QSQN-IDFS	361 (104+52+153+52)	154 (52+51+51)	204
		Magic-Sets	721 (212+103+302+104)	301 (103+98+100)	10105
		QSQR	410 (52+154+102+102)	358 (52+50+256)	356
	(b)	QSQN-IDFS	711 (204+102+303+102)	304 (102+101+101)	404
		Magic-Sets	1421 (412+203+602+204)	601 (203+198+200)	40205
		QSQR	810 (102+304+202+202)	708 (102+100+506)	706
Test 6.2	(a)	QSQN-IDFS	484 (113+104+211+56)	210 (55+100+55)	5207
		Magic-Sets	863 (229+158+366+110)	359 (107+148+104)	14082
		QSQR	433 (56+163+108+106)	377 (55+53+269)	3454
	(b)	QSQN-IDFS	934 (213+204+411+106)	410 (105+200+105)	20407
		Magic-Sets	1663 (429+308+716+210)	709 (207+298+204)	55657
		QSQR	833 (106+313+208+206)	727 (105+103+519)	11904
Test 6.3 (*i*)	(a)	QSQN-IDFS	39 (12+5+15+7)	16 (6+5+5)	2401
		Magic-Sets	30 (7+5+11+7)	14 (5+4+5)	2401
		QSQR	88 (12+24+28+24)	79 (12+9+58)	4403
	(b)	QSQN-IDFS	74 (22+10+30+12)	31 (11+10+10)	12751
		Magic-Sets	55 (12+10+21+12)	29 (10+9+10)	12751
		QSQR	168 (22+44+58+44)	149 (22+19+108)	24003
Test 6.3 (*ii*)	(a)	QSQN-IDFS	17 (3+5+7+2)	7 (1+5+1)	2001
		Magic-Sets	31 (10+7+10+4)	8 (2+4+2)	3521
		QSQR	50 (10+15+15+10)	35 (5+5+25)	2803
	(b)	QSQN-IDFS	27 (3+10+12+2)	12 (1+10+1)	11251
		Magic-Sets	41 (10+12+15+4)	13 (2+9+2)	20851
		QSQR	100 (20+30+30+20)	70 (10+10+50)	18003
Test 6.4	(a)	QSQN-IDFS	45 (3+19+21+2)	21 (1+19+1)	199
		Magic-Sets	61 (10+22+25+4)	23 (2+19+2)	853
		QSQR	190 (38+57+57+38)	133 (19+19+95)	348
	(b)	QSQN-IDFS	105 (3+49+51+2)	51 (1+49+1)	949
		Magic-Sets	121 (10+52+55+4)	53 (2+49+2)	4063
		QSQR	490 (98+147+147+98)	343 (49+49+245)	1848
Test 6.5		QSQN-IDFS	175 (7+54+59+55)	59 (3+53+3)	811
		Magic-Sets	181 (11+56+58+56)	58 (3+53+2)	792
		QSQR	675 (108+216+189+162)	537 (81+78+378)	3549
Test 6.6		QSQN-IDFS	60 (15+18+25+2)	21 (3+9+9)	1864
		Magic-Sets	159 (47+58+44+10)	41 (13+15+13)	3790
		QSQR	135 (25+50+50+10)	89 (15+9+65)	3020

Table 6.1: A comparison between QSQN, Magic-Sets and QSQR w.r.t. the number of read/write operations on relations and the maximum number of tuples/subqueries kept in the computer memory for the Experiments 1 and 2.

Methods	**Disk Reading** ($inp\_ + ans\_ + sup\_ + edb$) / **Disk Writing** ($inp\_ + ans\_ + sup\_$)		
Test 6.1 (a)	Memory limitation (corresponding to n_1, n_2, n_3 with m = 10105 tuples)		
	$n_1 \approx 50\%m$ (5052 tuples)	$n_2 \approx 30\%m$ (3031 tuples)	$n_3 \approx 20\%m$ (2021 tuples)
Strategy for unloading: *Timestamp*			
QSQN-IDFS	1 (0+0+0+1) / 0 (0+0+0)	1 (0+0+0+1) / 0 (0+0+0)	1 (0+0+0+1) / 0 (0+0+0)
Magic-Set	82 (26+1+26+29) / 54 (27+1+26)	262 (67+41+105+49) / 134 (47+41+46)	Not enough memory.
QSQR	1 (0+0+0+1) / 0 (0+0+0)	1 (0+0+0+1) / 0 (0+0+0)	1 (0+0+0+1) / 0 (0+0+0)
Strategy for unloading: *Extensional, Relation-size, Timestamp*			
QSQN-IDFS	1 (0+0+0+1) / 0 (0+0+0)	1 (0+0+0+1) / 0 (0+0+0)	1 (0+0+0+1) / 0 (0+0+0)
Magic-Set	60 (15+0+14+31) / 29 (15+0+14)	259 (65+41+102+51) / 129 (45+41+43)	Not enough memory.
QSQR	1 (0+0+0+1) / 0 (0+0+0)	1 (0+0+0+1) / 0 (0+0+0)	1 (0+0+0+1) / 0 (0+0+0)
Test 6.2 (a)	Memory limitation (corresponding to n_1, n_2, n_3 with m = 14082 tuples)		
	$n_1 \approx 50\%m$ (7041 tuples)	$n_2 \approx 30\%m$ (4224 tuples)	$n_3 \approx 20\%m$ (2816 tuples)
Strategy for unloading: *Timestamp*			
QSQN-IDFS	2 (0+0+0+2) / 0 (0+0+0)	4 (0+1+1+2) / 10 (3+2+5)	4 (0+1+1+2) / 11 (4+2+5)
Magic-Set	21 (6+1+5+9) / 16 (8+2+6)	154 (43+17+56+38) / 95 (38+20+37)	281 (71+45+113+52) / 153 (53+48+52)
QSQR	2 (0+0+0+2) / 0 (0+0+0)	2 (0+0+0+2) / 0 (0+0+0)	2 (0+0+0+2) / 58 (3+2+53)
Strategy for unloading: *Extensional, Relation-size, Timestamp*			
QSQN-IDFS	2 (0+0+0+2) / 0 (0+0+0)	4 (0+1+1+2) / 7 (3+1+3)	4 (0+1+1+2) / 10 (3+2+5)
Magic-Set	21 (5+1+3+12) / 12 (6+2+4)	139 (35+17+47+40) / 78 (30+20+28)	281 (70+45+111+55) / 149 (51+48+50)
QSQR	2 (0+0+0+2) / 0 (0+0+0)	2 (0+0+0+2) / 0 (0+0+0)	3 (0+0+1+2) / 6 (3+1+2)
Test 6.3 (i) (a)	Memory limitation (corresponding to n_1, n_2, n_3 with m = 4403 tuples)		
	$n_1 \approx 60\%m$ (2641 tuples)	$n_2 \approx 45\%m$ (1981 tuples)	$n_3 \approx 30\%m$ (1320 tuples)
Strategy for unloading: *Timestamp*			
QSQN-IDFS	1 (0+0+0+1) / 0 (0+0+0)	1 (0+0+0+1) / 1 (1+0+0)	7 (0+2+4+1) / 5 (1+3+1)
Magic-Set	1 (0+0+0+1) / 0 (0+0+0)	1 (0+0+0+1) / 1 (1+0+0)	7 (0+3+3+1) / 5 (1+3+1)
QSQR	7 (0+0+6+1) / 13 (2+0+11)	20 (2+3+13+2) / 24 (4+1+19)	43 (5+11+19+8) / 37 (7+2+28)
Strategy for unloading: *Extensional, Relation-size, Timestamp*			
QSQN-IDFS	1 (0+0+0+1) / 0 (0+0+0)	1 (0+0+0+1) / 1 (1+0+0)	6 (0+2+3+1) / 5 (1+3+1)
Magic-Set	1 (0+0+0+1) / 0 (0+0+0)	1 (0+0+0+1) / 1 (1+0+0)	7 (0+3+3+1) / 5 (1+3+1)
QSQR	7 (0+1+1+5) / 5 (1+1+3)	32 (3+6+10+13) / 23 (5+2+16)	49 (6+12+18+13) / 38 (8+3+27)
Test 6.3 (ii) (a)	Memory limitation (corresponding to n_1, n_2, n_3 with m = 3521 tuples)		
	$n_1 \approx 60\%m$ (2112 tuples)	$n_2 \approx 45\%m$ (1584 tuples)	$n_3 \approx 30\%m$ (1056 tuples)
Strategy for unloading: *Timestamp*			
QSQN-IDFS	1 (0+0+0+1) / 0 (0+0+0)	2 (0+0+1+1) / 3 (1+1+1)	Not enough memory.
Magic-Set	3 (1+0+1+1) / 4 (2+0+2)	4 (1+1+1+1) / 5 (2+1+2)	Not enough memory.
QSQR	9 (3+0+3+3) / 7 (3+0+4)	28 (5+9+7+7) / 17 (5+3+9)	Not enough memory.
Strategy for unloading: *Extensional, Relation-size, Timestamp*			
QSQN-IDFS	1 (0+0+0+1) / 0 (0+0+0)	2 (0+0+1+1) / 3 (1+1+1)	Not enough memory.
Magic-Set	3 (0+0+1+2) / 5 (2+1+2)	4 (1+0+1+2) / 5 (2+1+2)	Not enough memory.
QSQR	9 (1+1+2+5) / 6 (2+1+3)	23 (1+7+7+8) / 16 (4+3+9)	Not enough memory.

Table 6.2: A comparison between QSQN, Magic-Sets and QSQR for Experiment 1 w.r.t. the number of accesses to the secondary storage.

Methods	Disk Reading ($inp\_ + ans\_ + sup\_ + edb$) / Disk Writing ($inp\_ + ans\_ + sup\_$)		
Test 6.4 (a)	Memory limitation (corresponding to n_1, n_2, n_3 with $m = 853$ tuples)		
	$n_1 \approx 50\%m$ (426 tuples)	$n_2 \approx 30\%m$ (255 tuples)	$n_3 \approx 20\%m$ (170 tuples)
Strategy for unloading: *Timestamp*			
QSQN-IDFS	1 (0+0+0+1) / 0 (0+0+0)	1 (0+0+0+1) / 0 (0+0+0)	2 (0+0+1+1) / 3 (1+1+1)
Magic-Set	Not enough memory.	Not enough memory.	Not enough memory.
QSQR	1 (0+0+0+1) / 0 (0+0+0)	25 (3+7+7+8) / 17 (4+4+9)	48 (6+14+14+14) / 31 (8+7+16)
Strategy for unloading: *Extensional, Relation-size, Timestamp*			
QSQN-IDFS	1 (0+0+0+1) / 0 (0+0+0)	1 (0+0+0+1) / 0 (0+0+0)	2 (0+0+1+1) / 3 (1+1+1)
Magic-Set	Not enough memory.	Not enough memory.	Not enough memory.
QSQR	1 (0+0+0+1) / 0 (0+0+0)	21 (0+7+6+8) / 16 (4+4+8)	47 (6+14+13+14) / 29 (7+7+15)
Test 6.5	Memory limitation (corresponding to n_1, n_2, n_3 with $m = 3549$ tuples)		
	$n_1 \approx 50\%m$ (1774 tuples)	$n_2 \approx 30\%m$ (1064 tuples)	$n_3 \approx 20\%m$ (709 tuples)
Strategy for unloading: *Timestamp*			
QSQN-IDFS	3 (0+0+0+3) / 0 (0+0+0)	3 (0+0+0+3) / 0 (0+0+0)	4 (0+0+1+3) / 3 (2+0+1)
Magic-Set	3 (0+0+0+3) / 0 (0+0+0)	3 (0+0+0+3) / 0 (0+0+0)	4 (1+0+0+3) / 3 (2+0+1)
QSQR	55 (0+8+22+25) / 47 (15+8+24)	181 (10+39+60+72) / 123 (32+28+63)	Not enough memory.
Strategy for unloading: *Extensional, Relation-size, Timestamp*			
QSQN-IDFS	3 (0+0+0+3) / 0 (0+0+0)	3 (0+0+0+3) / 0 (0+0+0)	3 (0+0+0+3) / 0 (0+0+0)
Magic-Set	3 (0+0+0+3) / 0 (0+0+0)	3 (0+0+0+3) / 0 (0+0+0)	5 (0+0+0+5) / 1 (1+0+0)
QSQR	42 (0+5+10+27) / 23 (6+5+12)	161 (2+37+50+72) / 105 (26+26+53)	Not enough memory.
Test 6.6	Memory limitation (corresponding to n_1, n_2, n_3 with $m = 3790$ tuples)		
	$n_1 \approx 60\%m$ (2274 tuples)	$n_2 \approx 45\%m$ (1705 tuples)	$n_3 \approx 30\%m$ (1130 tuples)
Strategy for unloading: *Timestamp*			
QSQN-IDFS	1 (0+0+0+1) / 0 (0+0+0)	7 (1+3+2+1) / 11 (3+4+4)	9 (1+3+4+1) / 12 (3+4+5)
Magic-Set	26 (5+6+11+4) / 21 (8+7+6)	36 (6+8+17+5) / 25 (9+8+8)	51 (9+12+25+5) / 27 (10+9+8)
QSQR	2 (0+0+1+1) / 4 (2+0+2)	31 (4+10+12+5) / 33 (9+6+18)	33 (4+12+12+5) / 34 (9+7+18)
Strategy for unloading: *Extensional, Relation-size, Timestamp*			
QSQN-IDFS	1 (0+0+0+1) / 0 (0+0+0)	6 (0+3+2+1) / 11 (3+4+4)	7 (0+3+3+1) / 12 (3+4+5)
Magic-Set	20 (3+5+8+4) / 20 (7+7+6)	34 (5+8+16+5) / 25 (9+8+8)	52 (9+13+25+5) / 27 (10+9+8)
QSQR	2 (0+0+0+2) / 1 (1+0+0)	25 (0+8+12+5) / 24 (7+5+12)	27 (2+11+9+5) / 29 (7+7+15)

Table 6.3: A comparison between QSQN, Magic-Sets and QSQR for Experiment 2 w.r.t. the number of accesses to the secondary storage.

and 6.3 (on pages 78 and 79, respectively) show a comparison between the mentioned evaluation methods w.r.t. the number of accesses to the secondary storage.

In Table 6.1, the "Reading *inp_/ans_/sup_/edb*" column means the number of read operations on *input/answer/supplement/extensional* relations, respectively. Similarly, the "Writing *inp_/ans_/sup_*" column means the number of write operations on *input/answer/supplement* relations, respectively. The last column shows the maximum number of tuples/subqueries kept in the memory for each test.

Each of Tables 6.2 and 6.3 consists of four columns: the first one displays the names of methods, the next three columns show the numbers of read/write operations on the secondary storage when applying the mentioned limits n_1, n_2, n_3, respectively. Recall that the values n_1, n_2, n_3 are based on the value of $m = max\{m_1, m_2, m_3\}$, where m_1, m_2 and m_3 are the maximum numbers of tuples/subqueries kept in the memory for QSQN, QSQR and Magic-Sets, respectively, in the case when the memory is not restricted. A field with values "M $(M_1 + M_2 + M_3 + M_4)$ / N $(N_1 + N_2 + N_3)$" means: M (resp. M_1, M_2, M_3, M_4) is the total number of times of reading any (resp. *input*, *answer*, *supplement*, *extensional*) relation from the secondary storage, and N is the number of times of writing any (resp. *input*, *answer*, *supplement*) relation to the secondary storage. If there is more than one case in a test, we only show the results for the first case (i.e., the case (a)) in these tables. The results for all cases are provided online in [13].

As mentioned earlier, strategies for selecting an in-memory relation for unloading when there is not enough memory are based on the criteria *Timestamp, Relation-size, Extensional*. We used some of such strategies for our experiments, which are specified in the first row of each test.

There are also other tests for the comparison between the QSQN, Magic-Sets and QSQR methods, which are discussed in the next section and presented in Tables 6.4, 6.5 and 6.6 (on pages 85, 86 and 87, respectively).

As can be seen in Tables 6.1-6.6, the QSQR method is often worse than the QSQN and Magic-Sets methods w.r.t. the number of accesses to the secondary storage. As discussed in [39], QSQR uses iterative deepening search and clears input relations at the beginning of each iteration of the main loop, thus it allows redundant recomputations. In addition, the formulation of QSQR in [39] is at a logical level and uses the same relation for the whole sequence of supplements. This requires more relation loading/unloading when the recursive depth is high and no more memory is available.

The results in the mentioned tables also show that, there is not much difference between the QSQN and Magic-Sets methods w.r.t. the number of accesses to the secondary storage for queries with at least one bound parameter. For queries without any bound parameter, the number of accesses to the secondary storage in the case of the QSQN method is often a bit smaller than in the case of the Magic-Sets method. The reason is that, for a query like $p(x, y)$, the QSQN method stores only a fresh variant of (x, y) in *tuples*(*input_p*) because this is the most general tuple.

There are cases as in Tests 6.1 and 6.2 for which depth-first evaluation is more efficient than breadth-first evaluation and the QSQN-IDFS method as well as the QSQR method outperform the Magic-Sets method. See Tables 6.1 and 6.2 for more details on the experiments.

6.3 The QSQN-TRE method

This section presents our experimental results on the performance of the QSQN-TRE method in comparison with the QSQN method. The experimental measures are similar to the ones specified in Section 6.2. Additionally, we also compute the number of tuples and subqueries read from or written to the secondary storage for the mentioned methods when the memory is limited.

6.3.1 Experimental Settings

As mentioned before, the QSQN and QSQN-TRE methods allow any control strategies. In our tests, the IDFS control strategy is used for both QSQN and QSQN-TRE.

We also present the corresponding test results of the QSQR and Magic-Sets methods. A direct comparison between the QSQN-TRE method and the QSQR and Magic-Sets methods is somehow not appropriate because the former uses tail-recursion elimination, while the latter ones do not (we did not implement their variants that use tail-recursion elimination). The purpose of including the test results of the QSQR and Magic-Sets methods is to increase convenience for the reader in evaluating the efficiency of QSQN-TRE in a larger context.

The experimental settings are similar to the ones specified in Section 6.2.1, except that:

– For counting the maximum number of tuples/subqueries kept in the computer memory, we increase (resp. decrease) the counter by two if a tuple pair $(\bar{t}, \bar{t}')$ with $\bar{t} \neq \bar{t}'$ is added to (resp. removed from) a relation of the form $tuple\_pairs(input\_p)$, and increase (resp. decrease) it by one if a tuple, a subquery or a tuple pair of the form $(\bar{t}, \bar{t})$ is added to (resp. removed from) a relation.[6] The returned value is the maximum value of this counter.

– Regarding the second stage, the limit on the number of tuples/subqueries that can be kept in the computer memory is set as follows for each test. Let $M_{max} = max\{m_1, m_2, m_3, m_4\}$ and $M_{min} = min\{m_1, m_2, m_3, m_4\}$, where m_1, m_2, m_3, m_4 are the maximum numbers of tuples/subqueries kept in the memory when using QSQN-TRE, QSQN, QSQR, Magic-Sets, respectively, in the case when the memory is not restricted. Based on the value of M_{max} and M_{min}, we limit the maximal number of tuples/subqueries that can be kept in the memory sequentially to n_1, n_2 and n_3, where $n_1 \approx 60\% M_{max}$, $n_2 \approx 60\% n_1$ and $n_3 \approx 60\% M_{min}$. In order to avoid showing the monotonic results, in some tests, we use a different value for n_2, which is shown in the first row on each test. The user can change the limit before testing the package [13].

In order to make a comparison between the considered evaluation methods, we use the following tests:

Test 6.7. This test extends Test 6.1(b) by including the results of QSQN-TRE using $T(q_1) = T(q_2) = true$ and $T(p) = false$.

[6] A tuple pair of the form $(\bar{t}, \bar{t})$ can be encoded by the tuple $\bar{t}$ together with a Boolean flag.

Test 6.8. This test is taken from Example 4.1. We consider $T(p) = true$ (for QSQN-TRE) and the following values of m and n:

$$(a) \quad m = 20, n = 100; \qquad\qquad (b) \quad m = 100, n = 400.$$

As shown in Table 6.4 (on page 85), the maximum number of kept tuples in the case of using QSQN-TRE is much smaller than in the case of using QSQN or the other methods.

Test 6.9. This test is a modified version of the logic program given in Example 3.3, which is specified as follows:

$$\begin{aligned} &p(x, y) \leftarrow q(x, y)\\ &p(x, y) \leftarrow q(x, z), p(z, y)\\ &s(x, y) \leftarrow p(x, y) \end{aligned}$$

where p, s are intensional predicates with $T(p) = true$ and $T(s) = false$ (for QSQN-TRE), q is an extensional predicate, and x, y, z are variables. We consider this test with the following queries and extensional instance I for q:

a) The query is $s(a, x)$ and the extensional instance I for q is illustrated in Figure 6.1, where a and $a_{i,j}$ are constant symbols, using $m = 10$ and $n = 20$.

b) The query is $s(x, y)$ and the extensional instance I for q is specified as follows, using $n = 50$: $I(q) = \{(a_i, a_{i+1}) \mid 1 \leq i < n\} \cup \{(a_n, a_1)\}$.

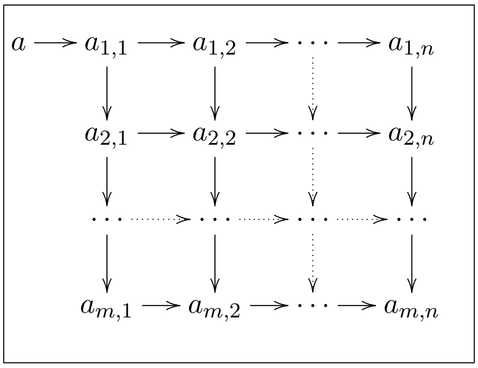


Fig. 6.1: A directed graph used for Test 6.9(a).

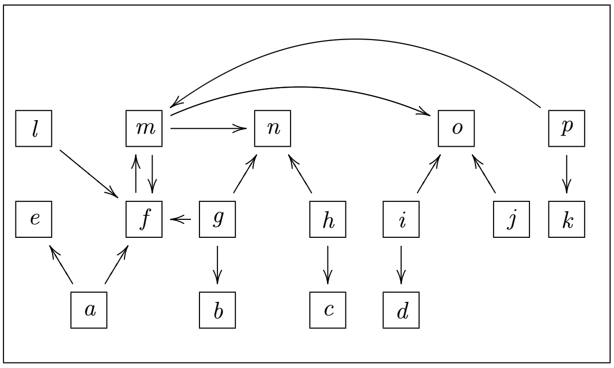


Fig. 6.2: The extensional instance used for Test 6.14.

Test 6.10. This test involves the transitive closure of a binary relation [8, 11]. It uses the following logic program P, where arc is an extensional predicate, $path$ is an intensional predicate with $T(path) = true$ (for QSQN-TRE), and x, y, z are variables.

$$\begin{aligned} &path(x, y) \leftarrow arc(x, y)\\ &path(x, y) \leftarrow path(x, z), path(z, y). \end{aligned}$$

The query is $path(a_0, x)$ and the extensional instance I for arc is specified in Figure 6.3, where a_0 and $a_{i,j}$ are constant symbols. We use $n = 5$ for this test.

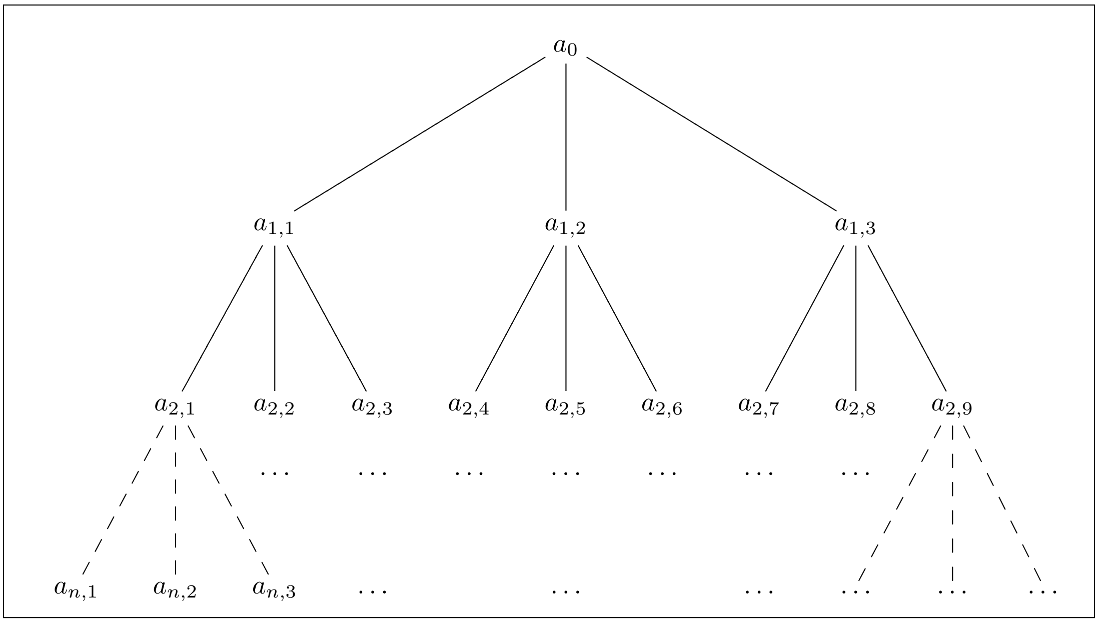


Fig. 6.3: The extensional instance used for Test 6.10.

Test 6.11. This test is taken from [22]. It uses the following "Same Generation Cousins" program, where *sgc* is an intensional predicate, *person*, *parent* are extensional predicates, w, x, y, z are variables.

$$sgc(x, x) \leftarrow person(x)$$
$$sgc(x, y) \leftarrow parent(x, z), sgc(z, w), parent(y, w).$$

The query is $sgc(oliver, x)$, where *oliver* is a constant symbol referring to a person's name. The data for this test were generated using generate_data website[7]. In particular, *person* has 150 tuples and *parent* has 350 tuples.

This is a non-tail recursive logic program. In this case, the QSQN-TRE evaluation method reduces to the QSQN method. As can be seen in Tables 6.4 and 6.6 (on pages 85 and 87, respectively) for this test, the QSQN-TRE and QSQN methods have the same experimental results.

6.3.2 Results and Discussion

By applying tail-recursion elimination, the QSQN-TRE method reduces materializing intermediate results during the processing. Our experiments show that, due to tail-recursion elimination, the QSQN-TRE method performs better than the QSQN method for a certain class of queries that depend on tail-recursive predicates.

Table 6.4 (on page 85) has the same style as Table 6.1. It shows a comparison between the QSQN-TRE, QSQN, QSQR and Magic-Sets evaluation methods w.r.t. the number of accesses to the intermediate relations and extensional relations as well

[7] http://www.generatedata.com

as the maximum number of kept tuples/subqueries in the memory for each of the tests. Tables 6.5 and 6.6 show a comparison between the mentioned evaluation methods w.r.t. the number of accesses to the secondary storage and the number of tuples/subqueries read from or written to the secondary storage. Each of these tables consists of the following columns: the first one displays the names of methods, the next three columns show the numbers of read/write operations on the secondary storage as well as the numbers of tuples and subqueries read from/written to the secondary storage when applying the mentioned limits n_1, n_2, n_3, respectively. A field with values "M $(M_1 + M_2 + M_3 + M_4)$ / N $(N_1 + N_2 + N_3)$" in the first line and "R / W" in the second line means: M (resp. M_1, M_2, M_3, M_4) is the total number of times of reading any (resp. *input*, *answer*, *supplement*, *extensional*) relation from the secondary storage, N (resp. N_1, N_2, N_3) is the total number of times of writing any (resp. *input*, *answer*, *supplement*) relation to the secondary storage, R (resp. W) is the number of tuples and subqueries read from (resp. written to) the secondary storage.

As shown by the results of Tests 6.7-6.9(a) in Tables 6.4 and 6.5, if the considered query depends on a tail-recursive predicate p such that:

- p occurs only in the last position in the bodies of the recursive clauses defining it,
- the adorned version of the logic program and the query uses only a unique adorned version of p, which has at least one bound parameter,[8]

then the QSQN-TRE method usually outperforms the other methods w.r.t.

- the number of read or write operations on relations,
- the maximum number of tuples and subqueries kept in the computer memory,
- the number of accesses to the secondary storage as well as the number of tuples and subqueries read from or written to the secondary storage when the memory is limited.

In contrast, for queries without any bound parameter as in Test 6.9(b) and for cases with a tail-recursive clause defining an intensional predicate p such that $T(p) = true$ and p occurs more then once in the clause's body as in Test 6.10, QSQN-TRE may be worse than QSQN. For Test 6.9(b), the reason is that, after processing a node $v = filter_{i,n_i}$ with $p = pred(v)$ and $T(p) = true$, QSQN-TRE produces a set of tuple pairs and accumulates them in $tuple\_pairs(input\_p)$, which are not instances of each other. Meanwhile, QSQN adds answers to $tuples(ans\_p)$ for later processing and transfers appropriate data through $(v, input\_p)$ without adding any new tuple to $tuples(input\_p)$ because it already contains a fresh variant of (x, y) that is more general than all the other tuples. Thus, in this case, QSQN-TRE may keep more tuples than QSQN. See the experimental results for Tests 6.9(b) and 6.10 in Tables 6.4 and 6.6 for more details.

As mentioned earlier, if there is no tail-recursion to eliminate or $T(p) = false$ for every intensional predicate p, the QSQN-TRE method reduces to the QSQN evaluation method. In this case, they have the same executions. For instance, as can be seen in Tables 6.4 and 6.6 for Test 6.11, the QSQN-TRE and QSQN methods return the same experimental results.

[8] For details about the adorned version of the logic program, see Appendix A.

Tests	Methods	Reading (times) *inp_/ans_/sup_/edb*	Writing (times) *inp_/ans_/sup_*	Max No. of kept tuples
Test 6.7	QSQN-TRE	512 (204+3+203+102)	205 (102+2+101)	405
	QSQN	711 (204+102+303+102)	304 (102+101+101)	404
	Magic-Sets	1421 (412+203+602+204)	601 (203+198+200)	40205
	QSQR	810 (102+304+202+202)	708 (102+100+506)	706
Test 6.8 (a)	QSQN-TRE	103 (41+1+40+21)	41 (20+1+20)	279
	QSQN	143 (41+21+60+21)	60 (20+20+20)	2160
	Magic-Sets	106 (22+21+42+21)	59 (19+20+20)	2160
	QSQR	123 (21+41+40+21)	121 (20+20+81)	4181
Test 6.8 (b)	QSQN-TRE	503 (201+1+200+101)	201 (100+1+100)	1199
	QSQN	703 (201+101+300+101)	300 (100+100+100)	40700
	Magic-Sets	506 (102+101+202+101)	299 (99+100+100)	40700
	QSQR	603 (101+201+200+101)	601 (100+100+401)	80801
Test 6.9 (a)	QSQN-TRE	157 (62+3+61+31)	63 (31+2+30)	1374
	QSQN	214 (62+31+90+31)	91 (31+30+30)	12695
	Magic-Sets	217 (66+32+88+31)	89 (31+29+29)	12694
	QSQR	540 (62+182+176+120)	422 (62+58+302)	33397
Test 6.9 (b)	QSQN-TRE	260 (103+3+103+51)	104 (51+2+51)	12453
	QSQN	115 (5+53+55+2)	55 (2+51+2)	5103
	Magic-Sets	130 (14+56+56+4)	56 (3+51+2)	7702
	QSQR	750 (150+250+250+100)	549 (100+99+350)	10105
Test 6.10	QSQN-TRE	82 (33+11+32+6)	32 (11+5+16)	9296
	QSQN	80 (26+16+32+6)	28 (6+8+14)	4373
	Magic-Sets	61 (16+22+16+7)	16 (6+5+5)	4009
	QSQR	78 (17+25+28+8)	56 (8+9+39)	7743
Test 6.11	QSQN-TRE	70 (18+9+25+18)	26 (9+9+8)	825
	QSQN	70 (18+9+25+18)	26 (9+9+8)	825
	Magic-Sets	55 (10+9+18+18)	24 (8+8+8)	825
	QSQR	138 (18+36+48+36)	138 (18+16+104)	1353

Table 6.4: A comparison between the QSQN-TRE, QSQN, QSQR and Magic-Sets methods w.r.t. the number of read/write operations on relations and the maximum number of tuples/subqueries kept in the computer memory.

Methods	Disk Reading ($inp\_ + ans\_ + sup\_ + edb$) / Disk Writing ($inp\_ + ans\_ + sup\_$) Number of tuples and subqueries Read from / Written to the secondary storage		
Test 6.7	Memory limitation (corresponding to n_1, n_2 and n_3)		
	$n_1 = 24123$ *tuples*	$n_2 = 14473$ *tuples*	$n_3 = 242$ *tuples*
Strategy for unloading: *Relation-size, Timestamp*			
QSQN-TRE	1 (0+0+0+1) / 0 (0+0+0) 100 / 0	1 (0+0+0+1) / 0 (0+0+0) 100 / 0	76 (1+0+20+55) / 75 (54+0+21) 15182 / 9882
QSQN	1 (0+0+0+1) / 0 (0+0+0) 100 / 0	1 (0+0+0+1) / 0 (0+0+0) 100 / 0	64 (31+0+1+32) / 32 (31+0+1) 5966 / 2766
Magic-Sets	65 (30+0+2+33) / 32 (31+0+1) 593630 / 273432	400 (109+60+148+83) / 199 (80+60+59) 3238911 / 1325782	Not enough memory.
QSQR	1 (0+0+0+1) / 0 (0+0+0) 100 / 0	1 (0+0+0+1) / 0 (0+0+0) 100 / 0	62 (30+0+0+32) / 31 (31+0+0) 5765 / 2666
Strategy for unloading: *Extensional, Relation-size, Timestamp*			
QSQN-TRE	1 (0+0+0+1) / 0 (0+0+0) 100 / 0	1 (0+0+0+1) / 0 (0+0+0) 100 / 0	97 (1+0+41+55) / 85 (43+0+42) 15103 / 9804
QSQN	1 (0+0+0+1) / 0 (0+0+0) 100 / 0	1 (0+0+0+1) / 0 (0+0+0) 100 / 0	63 (15+0+16+32) / 33 (17+0+16) 5781 / 2683
Magic-Sets	67 (17+0+16+34) / 34 (18+0+16) 576917 / 266619	389 (97+57+151+84) / 194 (69+57+68) 3143699 / 1291871	Not enough memory.
QSQR	1 (0+0+0+1) / 0 (0+0+0) 100 / 0	1 (0+0+0+1) / 0 (0+0+0) 100 / 0	97 (1+0+63+33) / 76 (3+0+73) 3435 / 247
Test 6.8 (a)	Memory limitation (corresponding to n_1, n_2 and n_3)		
	$n_1 = 2508$ *tuples*	$n_2 = 2002$ *tuples* ($\approx 80\% n_1$)	$n_3 = 167$ *tuples*
Strategy for unloading: *Relation-size, Timestamp*			
QSQN-TRE	2 (0+0+0+2) / 0 (0+0+0) 120 / 0	2 (0+0+0+2) / 0 (0+0+0) 120 / 0	2 (0+0+0+2) / 0 (0+0+0) 120 / 0
QSQN	2 (0+0+0+2) / 0 (0+0+0) 120 / 0	4 (0+1+1+2) / 3 (1+1+1) 2140 / 2040	Not enough memory.
Magic-Sets	2 (0+0+0+2) / 0 (0+0+0) 120 / 0	4 (0+1+1+2) / 3 (1+1+1) 2140 / 2040	Not enough memory.
QSQR	10 (0+8+0+2) / 24 (0+8+16) 12520 / 14000	13 (0+11+0+2) / 32 (1+11+20) 15520 / 17420	Not enough memory.
Strategy for unloading: *Extensional, Relation-size, Timestamp*			
QSQN-TRE	2 (0+0+0+2) / 0 (0+0+0) 120 / 0	2 (0+0+0+2) / 0 (0+0+0) 120 / 0	2 (0+0+0+2) / 0 (0+0+0) 120 / 0
QSQN	2 (0+0+0+2) / 0 (0+0+0) 120 / 0	4 (0+1+1+2) / 3 (1+1+1) 2140 / 2040	Not enough memory.
Magic-Sets	2 (0+0+0+2) / 0 (0+0+0) 120 / 0	4 (0+1+1+2) / 3 (1+1+1) 2140 / 2040	Not enough memory.
QSQR	10 (0+1+7+2) / 24 (1+1+22) 2027 / 3427	13 (0+2+9+2) / 32 (1+2+29) 3629 / 5529	Not enough memory.
Test 6.9 (a)	Memory limitation (corresponding to n_1, n_2 and n_3)		
	$n_1 = 20038$ *tuples*	$n_2 = 12022$ *tuples*	$n_3 = 824$ *tuples*
Strategy for unloading: *Relation-size, Timestamp*			
QSQN-TRE	1 (0+0+0+1) / 0 (0+0+0) 371 / 0	1 (0+0+0+1) / 0 (0+0+0) 371 / 0	12 (1+0+0+11) / 10 (10+0+0) 7561 / 3680
QSQN	1 (0+0+0+1) / 0 (0+0+0) 371 / 0	15 (0+7+7+1) / 8 (0+7+1) 83636 / 81039	Not enough memory.
Magic-Sets	1 (0+0+0+1) / 0 (0+0+0) 371 / 0	15 (0+7+7+1) / 8 (0+7+1) 83636 / 81039	Not enough memory.
QSQR	26 (0+15+10+1) / 26 (0+6+20) 174290 / 82655	147 (14+69+32+32) / 95 (16+14+65) 780985 / 153929	Not enough memory.
Strategy for unloading: *Extensional, Relation-size, Timestamp*			
QSQN-TRE	1 (0+0+0+1) / 0 (0+0+0) 371 / 0	1 (0+0+0+1) / 0 (0+0+0) 371 / 0	16 (0+0+5+11) / 12 (6+0+6) 6627 / 2747
QSQN	1 (0+0+0+1) / 0 (0+0+0) 371 / 0	2 (0+0+1+1) / 4 (2+0+2) 372 / 574	Not enough memory.
Magic-Sets	1 (0+0+0+1) / 0 (0+0+0) 371 / 0	1 (0+0+0+1) / 3 (2+0+1) 371 / 573	Not enough memory.
QSQR	64 (0+5+57+2) / 113 (4+3+106) 55578 / 44845	220 (27+53+99+41) / 192 (31+6+155) 625214 / 87655	Not enough memory.

Table 6.5: A comparison between QSQN-TRE, QSQN, QSQR and Magic-Sets for Tests 6.7-6.9(a) w.r.t. the number of accesses to the secondary storage as well as the number of tuples and subqueries read from/written to the secondary storage.

Methods	**Disk Reading** ($inp\_ + ans\_ + sup\_ + edb$) / **Disk Writing** ($inp\_ + ans\_ + sup\_$) **Number of tuples and subqueries Read from / Written to the secondary storage**		
Test 6.9 (b)	Memory limitation (corresponding to n_1, n_2 and n_3)		
	n_1 = 7471 *tuples*	n_2 = 4482 *tuples*	n_3 = 3061 *tuples*
Strategy for unloading: *Relation-size, Timestamp*			
QSQN-TRE	1 (0+0+0+1) / 2 (0+1+1) 50 / 5000	Not enough memory.	Not enough memory.
QSQN	1 (0+0+0+1) / 0 (0+0+0) 50 / 0	1 (0+0+0+1) / 1 (0+1+0) 50 / 2500	1 (0+0+0+1) / 1 (0+1+0) 50 / 2500
Magic-Sets	1 (0+0+0+1) / 1 (0+1+0) 50 / 2500	1 (0+0+0+1) / 2 (0+2+0) 50 / 5000	1 (0+0+0+1) / 2 (0+2+0) 50 / 5000
QSQR	17 (0+15+1+1) / 17 (0+15+2) 36100 / 38550	91 (0+63+27+1) / 79 (0+50+29) 181250 / 155500	158 (0+107+50+1) / 118 (0+66+52) 287300 / 209750
Strategy for unloading: *Extensional, Relation-size, Timestamp*			
QSQN-TRE	2 (0+0+1+1) / 4 (2+0+2) 51 / 7403	Not enough memory.	Not enough memory.
QSQN	1 (0+0+0+1) / 0 (0+0+0) 50 / 0	1 (0+0+0+1) / 5 (2+1+2) 50 / 2553	1 (0+0+0+1) / 5 (2+1+2) 50 / 2553
Magic-Sets	1 (0+0+0+1) / 6 (3+1+2) 50 / 2652	2 (1+0+0+1) / 7 (3+2+2) 51 / 5152	2 (1+0+0+1) / 7 (3+2+2) 51 / 5152
QSQR	41 (0+11+15+15) / 37 (10+11+16) 47600 / 51810	189 (6+59+75+49) / 145 (34+47+64) 208113 / 182291	306 (20+104+116+66) / 230 (56+65+109) 302991 / 224827
Test 6.10	Memory limitation (corresponding to n_1, n_2 and n_3)		
	n_1 = 5577 *tuples*	n_2 = 3346 *tuples*	n_3 = 2405 *tuples*
Strategy for unloading: *Relation-size, Timestamp*			
QSQN-TRE	3 (0+1+1+1) / 5 (1+1+3) 7412 / 10695	Not enough memory.	Not enough memory.
QSQN	1 (0+0+0+1) / 0 (0+0+0) 363 / 0	6 (0+3+2+1) / 4 (0+2+2) 8082 / 6078	12 (2+4+5+1) / 8 (1+3+4) 10984 / 8252
Magic-Sets	1 (0+0+0+1) / 0 (0+0+0) 363 / 0	6 (0+3+2+1) / 2 (0+1+1) 8568 / 3282	9 (0+4+4+1) / 3 (0+1+2) 11061 / 3708
QSQR	1 (0+0+0+1) / 1 (0+0+1) 363 / 1728	9 (1+5+1+2) / 7 (2+2+3) 10213 / 7934	15 (2+7+4+2) / 12 (2+3+7) 14222 / 10262
Strategy for unloading: *Extensional, Relation-size, Timestamp*			
QSQN-TRE	3 (0+1+0+2) / 4 (1+1+2) 6013 / 8933	Not enough memory.	Not enough memory.
QSQN	1 (0+0+0+1) / 0 (0+0+0) 363 / 0	6 (2+1+2+1) / 4 (1+1+2) 4737 / 4010	9 (2+3+3+1) / 6 (1+2+3) 9174 / 6806
Magic-Sets	1 (0+0+0+1) / 0 (0+0+0) 363 / 0	9 (2+3+2+2) / 4 (2+1+1) 9416 / 3767	9 (2+3+2+2) / 4 (2+1+1) 9416 / 3767
QSQR	1 (0+0+0+1) / 0 (0+0+0) 363 / 0	5 (0+2+1+2) / 6 (1+1+4) 4371 / 4459	15 (1+5+6+3) / 15 (2+2+11) 11662 / 9626
Test 6.11	Memory limitation (corresponding to n_1, n_2 and n_3)		
	$n_1 = 811$ *tuples*	$n_2 = \lfloor (n_1 + n_3)/2 \rfloor = 653$ *tuples*	n_3 = 495 *tuples*
Strategy for unloading: *Relation-size, Timestamp*			
QSQN-TRE	2 (0+0+0+2) / 0 (0+0+0) 500 / 0	6 (0+2+0+4) / 2 (0+2+0) 1681 / 481	9 (0+3+0+6) / 3 (0+3+0) 2517 / 617
QSQN	2 (0+0+0+2) / 0 (0+0+0) 500 / 0	6 (0+2+0+4) / 2 (0+2+0) 1681 / 481	9 (0+3+0+6) / 3 (0+3+0) 2517 / 617
Magic-Sets	2 (0+0+0+2) / 0 (0+0+0) 500 / 0	6 (0+2+0+4) / 2 (0+2+0) 1681 / 481	9 (0+3+0+6) / 3 (0+3+0) 2517 / 617
QSQR	37 (0+17+0+20) / 2 (0+2+0) 11923 / 503	40 (0+18+0+22) / 2 (0+2+0) 12731 / 503	72 (0+20+5+47) / 12 (0+4+8) 18864 / 1322
Strategy for unloading: *Extensional, Relation-size, Timestamp*			
QSQN-TRE	2 (0+0+0+2) / 0 (0+0+0) 500 / 0	3 (0+0+0+3) / 0 (0+0+0) 850 / 0	9 (0+2+1+6) / 4 (1+2+1) 2389 / 498
QSQN	2 (0+0+0+2) / 0 (0+0+0) 500 / 0	3 (0+0+0+3) / 0 (0+0+0) 850 / 0	9 (0+2+1+6) / 4 (1+2+1) 2389 / 498
Magic-Sets	2 (0+0+0+2) / 0 (0+0+0) 500 / 0	3 (0+0+0+3) / 0 (0+0+0) 850 / 0	9 (0+2+1+6) / 4 (1+2+1) 2389 / 498
QSQR	26 (0+2+2+22) / 19 (1+2+16) 6450 / 684	64 (7+18+9+30) / 35 (9+2+24) 13975 / 767	99 (7+19+26+47) / 53 (9+3+41) 18648 / 1090

Table 6.6: A comparison between QSQN-TRE, QSQN, QSQR and Magic-Sets for Tests 6.9(b)-6.11 w.r.t. the number of accesses to the secondary storage as well as the number of tuples and subqueries read from/written to the secondary storage.

6.4 The QSQN-rTRE method

This section presents our experimental results related to the efficiency of the QSQN-rTRE method in comparison with the QSQN-TRE method. The comparison is made with respect to the number of read/write operations on relations as well as the maximum number of kept tuples/subqueries in the computer memory.

6.4.1 Experimental Settings

As mentioned earlier, the QSQN-rTRE method allows various control strategy. In our tests for QSQN-rTRE, we use the IDFS control strategy. We assume that the computer memory is large enough to load all the involved relations. During the processing, for each operation of reading from a relation (resp. writing a set of tuples to a relation), we increase the counter of read (resp. write) operations on this relation by one. For counting the maximum number of tuples/subqueries kept in the computer memory, regarding the QSQN-TRE method, we increase (resp. decrease) the counter by two if a tuple pair $(\bar{t}, \bar{t}')$ with $\bar{t} \neq \bar{t}'$ is added to (resp. removed from) a relation of the form *tuple_pairs*(*input_p*), and increase (resp. decrease) it by one if a tuple, a subquery or a tuple pair of the form $(\bar{t}, \bar{t})$ is added to (resp. removed from) a relation. For the QSQN-rTRE method, we increase (resp. decrease) the counter of kept tuples by two if a tuple-atom pair is added to (resp. removed from) *ta_pairs*(*input_p*), otherwise we increase (resp. decrease) it by one. The returned value is the maximum value of this counter. We consider the following tests for the comparison.

Test 6.12. This test uses the program P given in Example 4.3 with $T(q) = true$, $T(p) = T(s) = false$ for QSQN-TRE and $T(q) = T(p) = T(s) = true$ for QSQN-rTRE. The query is $s(x)$ and the extensional instance I for t is as follows, where a, a_i, b_i are constant symbols:

$$\begin{aligned} I(t) = \; & \{(a, a_1)\} \;\cup\; \{(a_i, a_{i+1}) \mid 1 \leq i < n\} \;\cup\; \{(a_n, a_1)\} \;\cup \\ & \{(a, b_1)\} \;\cup\; \{(b_i, b_{i+1}) \;\mid 1 \leq i < n\}. \end{aligned}$$

We perform this test using the following values of n:

$$(a)\;\; n = 100, \qquad (b)\;\; n = 500, \qquad (c)\;\; n = 1000.$$

Test 6.13. This test uses the following logic program P, where p, q are intensional predicates, t_1, t_2 are extensional predicates, and x, y, z are variables. In this test, we consider the case when p and q are mutually dependent on each other with $T(p) = T(q) = false$ for QSQN-TRE and $T(p) = T(q) = true$ for QSQN-rTRE.

$$\begin{aligned} p(x, y) &\leftarrow t_1(x, y) \\ p(x, y) &\leftarrow t_1(x, z), q(z, y) \\ q(x, y) &\leftarrow t_2(x, y) \\ q(x, y) &\leftarrow t_2(x, z), p(z, y). \end{aligned}$$

Tests		Methods	Reading (times) $inp\_/ans\_/sup\_/edb$	Writing (times) $inp\_/ans\_/sup\_$	Max No. of kept tuples
Test 6.12	(a)	QSQN-TRE	520 (206+5+207+102)	208 (103+3+102)	1406
		QSQN-rTRE	514 (206+1+205+102)	206 (103+1+102)	1008
	(b)	QSQN-TRE	2520 (1006+5+1007+502)	1008 (503+3+502)	7006
		QSQN-rTRE	2514 (1006+1+1005+502)	1006 (503+1+502)	5008
	(c)	QSQN-TRE	5020 (2006+5+2007+1002)	2008 (1003+3+1002)	14006
		QSQN-rTRE	5014 (2006+1+2005+1002)	2006 (1003+1+1002)	10008
Test 6.13	(a)	QSQN-TRE	704 (201+103+298+102)	300 (100+101+99)	5248
		QSQN-rTRE	503 (201+2+198+102)	201 (100+2+99)	497
	(b)	QSQN-TRE	1404 (401+203+598+202)	600 (200+201+199)	20498
		QSQN-rTRE	1003 (401+2+398+202)	401 (200+2+199)	997
	(c)	QSQN-TRE	2104 (601+303+898+302)	900 (300+301+299)	45748
		QSQN-rTRE	1503 (601+2+598+302)	601 (300+2+299)	1497
Test 6.14	(a)	QSQN-TRE	25 (7+3+9+6)	9 (3+3+3)	60
		QSQN-rTRE	25 (7+3+9+6)	9 (3+3+3)	60
	(b)	QSQN-TRE	15 (3+3+5+4)	5 (1+3+1)	36
		QSQN-rTRE	15 (3+3+5+4)	5 (1+3+1)	36

Table 6.7: A comparison between the QSQN-TRE and QSQN-rTRE methods w.r.t. the number of read/write operations on relations and the maximum number of tuples/subqueries kept in the computer memory.

The query is $q(a_1, x)$ and the extensional instance I for t_1 and t_2 is as follows, where a_i $(1 \leq i \leq n)$ are constant symbols:

$$\begin{aligned} I(t_1) = &\quad \{(a_2, a_3), (a_4, a_5), (a_6, a_7), \ldots, (a_{n-2}, a_{n-1})\}, \\ I(t_2) = &\quad \{(a_1, a_2), (a_3, a_4), (a_5, a_6), \ldots, (a_{n-1}, a_n)\}. \end{aligned}$$

We perform this test using the following values of n:

$$(a)\quad n = 100, \qquad (b)\quad n = 200, \qquad (c)\quad n = 300.$$

Test 6.14. This test is taken from [1]. It uses the following "Reverse-Same-Generation" (RSG) program, where rsg is an intensional predicate, $flat$, up, $down$ are extensional predicates, x, y, z, w are variables:

$$\begin{aligned} &rsg(x, y) \leftarrow flat(x, y) \\ &rsg(x, y) \leftarrow up(x, z), rsg(w, z), down(w, y). \end{aligned}$$

We use a small dataset for this test, which is illustrated in Figure 6.2, where $a - p$ are constant symbols. We perform this test using the following queries:

$$(a)\quad rsg(a, x), \qquad\qquad (b)\quad rsg(x, y).$$

This is a non-right/tail recursive logic program. As can be seen in Table 6.7, the QSQN-rTRE and QSQN-TRE methods have the same experimental results.

6.4.2 Results and Discussion

Table 6.7 shows the comparison between the QSQN-TRE and QSQN-rTRE evaluation methods w.r.t. the number of accesses to the relations as well as the maximum number of kept tuples/subqueries in the computer memory. In this table, the third column means the number of read operations on *input/answer/supplement/extensional* relations, respectively. Similarly, the fourth column means the number of write operations on *input/answer/supplement* relations, respectively. The last column shows the maximum number of kept tuples in the computer memory. As can be seen in this table, by not representing intermediate results during the computation for the right/tail-recursive cases, the QSQN-rTRE method usually outperforms the QSQN-TRE method for a certain class of queries that depends on right/tail-recursive predicates with respect to the number of accesses to the relations as well as the maximum number of kept tuples/subqueries in the computer memory. Especially, for the case when the intensional predicates are mutually (rightmost) dependent on each other as in Test 6.13, the maximum number of kept tuples in the case of using QSQN-rTRE is much smaller than in the case of using QSQN-TRE.

6.5 The QSQN-STR method

In this section, we present the experimental results of the QSQN-STR method and discuss its performance. We make a comparison between the QSQN-STR method and Datalog Educational System (DES [56], a deductive database system) with respect to the number of generated tuples in answer relations corresponding to the intensional predicates. We begin with the following definition.

Definition 6.2. The *global-priority* of an active edge (v, w) in a QSQN-STR, where v is of the form $input\_p$, ans_p or $filter_{i,j}$ for some p, i and j, is a vector $global\text{-}priority(v, w) = (a, b)$, where a and b are defined as follows:

- $a = 2$ if $v = input\_p$, $a = 1$ if $v = filter_{i,j}$, and $a = 0$ if $v = ans\_p$,
- b is the priority of (v, w) specified in Definition 6.1 if $v = input\_p$ or $v = ans\_p$, and b is the modification timestamp of v if $v = filter_{i,j}$. ∎

In order to satisfy the admissibility w.r.t. strata's stability, we use a slightly modified version of the IDFS control strategy, called IDFS2. This strategy differs from IDFS (presented on page 69) at Steps 1 and 3. Particularly, the initial values for $input\_q$ and the relations of the form $unprocessed(input\_q, v)$ used for QSQN-STR are set in the same way as for QSQN at Step 1. The modification for Step 3 is as follows:

While the stack is not empty do:

(a) pop an edge (u, v) from the stack,

(b) if (u, v) is an "active" edge then (b.1) if $u = ans\_p$, $v = filter_{i,j}$, $pred(v) = p$ and p is not the predicate of A_i then[9]

(b.1.1) if $layer(u) = layer(v)$ and there exist active edges (u', v') with $u' = input\_p$ then[10]
- push (u, v) into the stack,
- set (u, v) to be the edge with the highest priority from those active edges (u', v'),

(b.1.2) else if $layer(u) < layer(v)$ and there exist active edges (u', v') such that $layer(u') \leq layer(u)$ and $layer(v') \leq layer(u)$ then[11]
- push (u, v) into the stack,
- set (u, v) to be the edge with the highest global-priority from those active edges (u', v'),

(b.1.3) else if $layer(u) < layer(v)$ and there exist active edges $(u', v') \neq (u, v)$ such that $layer(u') \leq layer(u)$ and $layer(v') > layer(u)$ then[12]
- set $(u, v) = (u'', v'')$, where (u'', v'') is the edge with the highest global-priority from those active edges (u', v') with the properties that $layer(v'') = h$ and h is the smallest layer number such that $h > layer(u)$ and $subqueries(v'') \neq \emptyset$,
- for each (u^*, v^*) from the remaining edges among those active edges (u', v') in the increasing order w.r.t. their global-priorities do
  - $*$ if $subqueries(v^*) \neq \emptyset$ then
    - "fire" the edge (u^*, v^*),
    - push $(v^*, succ(v^*))$ into the stack,
  - $*$ else set $unprocessed(u^*, v^*) = \emptyset$,

(b.2) "fire" the edge (u, v),

(b.3) push all the "active" edges outcoming from v into the stack in the increasing order w.r.t. their priorities,

(b.4) if $v = filter_{i,j}$, $pred(v) = p$, the predicate of A_i is p, the edge $(v, succ_2(v))$ is not "active" and there exist active edges (u', v') with $u' = input\_p$ then[13].
- push (u', v'') into the stack, where (u', v'') is the edge with the highest priority from those active edges (u', v'),

6.5.1 Experimental Settings

The number of generated tuples in an answer relation of QSQN-STR is defined to be the maximum number of tuples that were added to that relation. For counting the number of generated tuples in answer relations for each below test using DES, we use

[9]We have that $layer(u) \leq layer(v)$.

[10]In this case, the predicate of A_i and p are mutually dependent on each other.

[11]The goal of Steps $(b.1.2)$ and $(b.1.3)$ is to satisfy the admissibility w.r.t. strata's stability before turning to a higher layer.

[12]Before turning to a higher layer, "fire" all remaining active edges of the form (ans_p_k, v') in the increasing order w.r.t. their global-priorities, for some p_k such that $layer(ans\_p_k) \leq layer(u)$ and $layer(v') > layer(u)$.

[13]The aim is to accumulate as many as possible tuples in $tuples(ans\_p)$ before processing it.

the following commands:

$$/\mathit{consult} \quad <\mathit{file\_name}>, \text{ for consulting a program, and}$$
$$/\mathit{trace\_datalog} \quad <\mathit{a\_query}>, \text{ for tracing a query.}$$

The following tests are used for the comparison between the QSQN-STR method and DES. Regarding the QSQN-STR method, we assume that $T(v) = \mathit{false}$ for each $v = \mathit{filter}_{i,j} \in V$ with $\mathit{pred}(v) = \mathit{extensional}$.

Test 6.15. This test is taken from Example 5.1, the query is $\mathit{acyclic}(a, x)$ and the extensional instance I for edge is as follows, where a, a_i, b_i, c_i, d_i are constant symbols and $n = 50$:

$$\begin{aligned} I(\mathit{edge}) = \;& \{(a, a_1)\} \;\cup\; \{(a_i, a_{i+1}) \mid 1 \le i < n\} \;\cup\; \{(a_n, a_1)\} \cup \\ & \{(a, b_1)\} \;\cup\; \{(b_i, b_{i+1}) \;\mid 1 \le i < n\} \;\cup\; \{(b_n, b_1)\} \cup \\ & \{(c_i, c_{i+1}), (d_i, d_{i+1}) \mid 1 \le i < n\} \;\cup\; \{(c_n, c_1), (d_n, d_1)\}. \end{aligned}$$

Test 6.16. This test uses a semi-positive program taken from [32]. It computes pairs of nodes (x, y) such that y is reachable from x but not directly linked from x. In this program, which is specified below, *reachable* and *indirect* are intensional predicates, *link* is an extensional predicate, x, y and z are variables.

$$\mathit{reachable}(x, y) \leftarrow \mathit{link}(x, y)$$
$$\mathit{reachable}(x, y) \leftarrow \mathit{link}(x, z), \mathit{reachable}(z, y)$$
$$\mathit{indirect}(x, y) \leftarrow \mathit{reachable}(x, y), \sim\mathit{link}(x, y).$$

Let the query be $\mathit{indirect}(a, x)$ and the extensional instance I for *link* be as follows, where a, a_i, b_i are constant symbols and $n = 50$:

$$\begin{aligned} I(\mathit{link}) = \;& \{(a, a_1)\} \;\cup\; \{(a_i, a_{i+1}) \mid 1 \le i < n\} \;\cup\; \{(a_n, a_1)\} \cup \\ & \{(b_i, b_{i+1}) \mid 1 \le i < n\} \;\cup\; \{(b_n, b_1)\}. \end{aligned}$$

Test 6.17. This test uses the "cousins at the same generation" program specified below, which is a modified version of a program in [68]. It is a non-recursive program, where *sibling*, *grandparent* and *cousin* are intensional predicates, *parent* is an extensional predicate, x, y, z are variables:

$$\mathit{sibling}(x, y) \leftarrow \mathit{parent}(z, x), \mathit{parent}(z, y), x \mathbin{\backslash=} y$$
$$\mathit{grandparent}(x, y) \leftarrow \mathit{parent}(x, z), \mathit{parent}(z, y)$$
$$\mathit{cousin}(x, y) \leftarrow \mathit{grandparent}(z, x), \mathit{grandparent}(z, y), \sim\mathit{sibling}(x, y), x \mathbin{\backslash=} y.$$

In addition to the extensional and intensional predicates, our implementation can deal with some arithmetic operators. In this program, we use the predicate \=, where $(x \mathbin{\backslash=} y)$ denotes $\sim(x == y)$ with the meaning that x and y are not the same. For

Intensional relations	Generated tuples	
	DES	QSQN-STR
Test 6.15		
path	10200	5100
acyclic	100	100
Test 6.16		
reachable	2550	2550
indirect	49	49
Test 6.17		
sibling	1597	544
grandparent	354	354
cousin	702	702

Intensional relations	Generated tuples	
	DES	QSQN-STR
Test 6.18		
node	101	101
reachable	5101	2550
unreachable	51	51
Test 6.19		
q_1	466	30
q_2	13922	0
p	1	1

Table 6.8: A comparison between QSQN-STR and DES w.r.t. the number of the generated tuples in answer relations corresponding to the intensional predicates.

instance, the first clause says that x and y are siblings if they have the same parents and x is not y (i.e., they are not the same individual).

The query for this test is $cousin(x, y)$ and the extensional instance I for *parent* was generated using generate_data website[14], which contains 350 tuples.

Test 6.18. This test computes all pairs of disconnected nodes in a graph. It is taken from [32], where *reachable*, *node* and *unreachable* are intensional predicates, *link* is an extensional predicate, x, y and z are variables. The query is $unreachable(a, x)$ and the extensional instance I for *link* is the same as in Test 6.16 using $n = 50$. The program is specified as follows:

$$reachable(x, y) \leftarrow link(x, y)$$
$$reachable(x, y) \leftarrow link(x, z), reachable(z, y)$$
$$node(x) \leftarrow link(x, y)$$
$$node(y) \leftarrow link(x, y)$$
$$unreachable(x, y) \leftarrow node(x), node(y), \sim reachable(x, y).$$

Test 6.19. The program, the extensional instance and the query for this test are taken from Example 5.2 using $m = n = 30$.

6.5.2 Results and Discussion

Table 6.8 shows the comparison between QSQN-STR and DES [56] w.r.t. the maximum number of generated tuples in answer relations corresponding to the intensional

[14] http://www.generatedata.com

predicates for QSQN-STR and the number of generated tuples in answer relations corresponding to the intensional predicates for DES. As can be seen in this table, QSQN-STR and DES have the same results in intensional relations for the semi-positive program in Test 6.16. However, the number of generated tuples in the answer relations corresponding to negated intensional predicates for QSQN-STR is often smaller than DES. Due to the top-down approach of IDFS2, at the time of processing a negative literal, the corresponding subquery contains no variables. Thus, this takes the advantage of reducing the number of generated tuples in relations corresponding to negated intensional predicates.

Chapter 7

Conclusions

The Horn fragment of first-order logic plays an important role in knowledge representation and reasoning. It is used as the language of definite logic programs and goals in logic programming. Its range-restricted and function-free version is also used as the Datalog language for deductive databases. Recently, rule-based query languages, including languages related to Datalog, have drawn a great deal of attention from researchers, especially as rule languages are now applied in areas such as the Semantic Web.

7.1 Summary of Contributions

We have formulated the first framework for developing algorithms for evaluating queries to Horn knowledge bases with the properties that: the approach is goal-directed; each subquery is processed only once and each supplement tuple, if desired[1], is transferred only once; operations are done set-at-a-time; and any control strategy can be used.

Our framework is an adaptation and a generalization of the QSQ approach of Datalog for Horn knowledge bases. One of the key differences is that we do not use adornments and annotations, but use substitutions instead. This is natural for the case with function symbols and without the range-restrictedness condition. When restricting to Datalog queries, it groups operations on the same relation together regardless of adornments and allows to reduce the number of accesses to the secondary storage although "joins" would be more complicated.

QSQ-nets are a more intuitive representation than the description of the QSQ approach of Datalog given in [1]. Particularly, we transform a logic program into an equivalent net structure and use it to determine which set of tuples or subqueries should be evaluated at each step, in an efficient way. Our notion of QSQ-net makes a connection to flow networks and is intuitive for developing efficient evaluation algorithms. For example, as shown in Chapter 4, it is easy to incorporate tail-recursion elimination and right/tail-recursion elimination into QSQ-nets.

Our framework forms a generic evaluation method called QSQN. This method is designed so that the query processing is divided into appropriate steps which can be delayed to maximize adjustability and allow various control strategies. In comparison

[1]when $T(v) = \mathit{false}$ for all nodes v of the form $\mathit{filter}_{i,j}$ with $\mathit{kind}(v) = \mathit{extensional}$.

with the most well-known evaluation methods, the generic QSQN evaluation method does not do redundant recomputations as the QSQR evaluation method and is more adjustable and thus has essential advantages over the Magic-Sets evaluation method.

The QSQN method is much different from the QSQR method. Despite that both the QSQN evaluation method proposed in this dissertation and the QSQR method proposed in [39] deal with query processing for Horn knowledge bases, they are fundamentally different:

- As discussed in [39, Remark 3.2], QSQR uses iterative deepening search and clears input relations at the beginning of each iteration of the main loop, and thus allows redundant recomputations. In contrast, QSQN allows any control strategies and reduces redundant recomputations.
- Using the recursive approach, a path of recursive calls by QSQR may be long and involves a considerable number of relations. When no more computer memory is available, this causes many operations of loading/unloading relations from/to the secondary storage. In contrast, the processing in QSQN is divided into smaller steps and QSQN has the adjustability in choosing an operation for the next step. This allows accumulating tuples/subqueries at each node of the net before processing them together (set-at-a-time), and hence reduces the number of accesses to the secondary storage.

The QSQN method is sound and complete, and when the term-depth bound is fixed, it has polynomial time data complexity. Notice the significance of this: it states that one can develop and use any control strategy for QSQN and the resulting evaluation method is always guaranteed to be sound and complete. The properties on soundness, completeness and data complexity of QSQN are important in the context that, without proofs, the methods proposed in [1, 38, 69] were wrongly claimed to be complete.

We evaluated the usefulness of the generic QSQN evaluation method as follows:

- We proposed three control strategies DAR, DFS, IDFS and implemented QSQN together with these strategies to obtain the corresponding evaluation methods QSQN-DAR, QSQN-DFS and QSQN-IDFS. The intention of DAR is to reduce the number of accesses to the secondary storage. However, our current implementation of the DAR control strategy is not advanced enough and the implemented QSQN-DAR method is not more efficient than the implemented QSQN-IDFS method. So, for comparison with the Magic-Sets and QSQR methods we used QSQN-IDFS.
- We also implemented the Magic-Sets and QSQR methods for the comparison.
- We compared the implemented QSQN-IDFS, QSQR and Magic-Sets methods using representative examples that appeared in well-known articles on deductive databases as well as new examples. The comparison was made w.r.t. the following measures:
  - the number of read/write operations on relations,
  - the maximum number of tuples/subqueries kept in the computer memory for the case when the memory is large enough to hold all the related extensional relations as well as the intermediate relations,

- the number of accesses to the secondary storage when the memory is limited.

We chose these measures because they (essentially) affect the execution time but not vice versa, while the execution time is also affected by various optimization techniques as well as by data management at the physical level. Comparison results on the execution time and the number of tuples/subqueries read from or written to the secondary storage are less representative and provided only online in [13].

Our experiments in Section 6.2 show that the QSQN-IDFS evaluation method is more efficient than the QSQR evaluation method and as competitive as the Magic-Sets evaluation method. In the case when the order of program clauses and the order of atoms in the bodies of program clauses are essential as in Prolog programming, the QSQN-IDFS evaluation method usually outperforms the Magic-Sets method. As QSQN-IDFS is just an instance of the generic QSQN evaluation method, we conclude that this generic method is useful.

We have incorporated tail-recursion elimination into query-subquery nets in order to formulate the QSQN-TRE evaluation method for Horn knowledge bases, which allows to reduce materializing intermediate results during the processing. We have proved soundness and completeness of the QSQN-TRE evaluation method and showed that, when the term-depth bound is fixed, the method has polynomial time data complexity. Similarly to QSQN, our new method allows various control strategies such as DAR, DFS and IDFS. The experimental results in Section 6.3 show that, if the considered query depends on a tail-recursive predicate p such that:

- p occurs only in the last position in the bodies of the recursive clauses defining it,
- the adorned version of the logic program and the query uses only a unique adorned version of p, which has at least one bound parameter,

then the QSQN-TRE method usually outperforms the other methods w.r.t.

- the number of read or write operations on relations,
- the maximum number of tuples and subqueries kept in the computer memory,
- the number of accesses to the secondary storage as well as the number of tuples and subqueries read from or written to the secondary storage when the computer memory is limited.

Additionally, we have proposed another method called QSQN-rTRE for evaluating queries to Horn knowledge bases, which can eliminate not only tail-recursive predicates proposed in Section 4.1, but also intensional predicates that appear rightmost in the bodies of the program clauses. The aim is to reduce materializing intermediate results for a certain class of queries that depends on right/tail-recursive predicates. Especially, for the case when the intensional predicates are mutually (rightmost) dependent on each other as in Test 6.13. The usefulness of this method is illustrated by empirical results in Section 6.4.

Besides, we have incorporated stratified negation into query-subquery nets to obtain the QSQN-STR method for evaluating queries to stratified knowledge bases. We have proved the soundness and completeness of QSQN-STR for the case without function symbols. The experimental results in Section 6.5 indicate the usefulness of this method.

7.2 Future Work

As discussed in the previous chapters, although the provided methods for evaluating queries to Horn knowledge bases or stratified knowledge bases are useful, there are some improvements that could still be made to improve the performance of the QSQN method and its extensions. This section briefly describes some interesting research topics, which are worth investigating further. In the future, our work will mainly concentrate on the following tasks:

- As mentioned earlier, QSQN and its extensions allow various control strategies, we will develop better control strategies for QSQN, which focus on how to reduce the number of accesses to the secondary storage as much as possible.
- A possible work would be to develop optimization techniques for QSQN in processing Datalog queries by using adornments.
- Other directions would be to incorporate into QSQN the optimization techniques proposed in [42, 48, 61, 65] as well as apply our evaluation methods to the Datalog-like rule languages for the Semantic Web proposed in our previous works [18, 19, 20, 21].
- Another area of interest for the application of the proposed methods is in working with large datasets [25, 66].
- We will implement a variant of the Magic-Sets method without adornments and extend our comparison to also cover that modified method. This will make a comprehensive comparison between the QSQN and Magic-Sets methods.
- It is desirable to consider normal logic programs, which allow negation to occur in the bodies of program clauses. Therefore, we will extend the QSQN-STR method for dealing with this language using the well-founded semantics [23, 34, 64].

Appendix A

Existing Methods for Query Evaluation

Researchers have developed a number of evaluation methods for Datalog deductive databases or Horn knowledge bases such as QSQ [1, 69], QSQR [43, 69], QoSaQ [71] and Magic-Sets [1, 7, 8]. These evaluation methods have both advantages and disadvantages. In this appendix, we present in brief the most well-known methods for evaluating queries to Datalog deductive databases or Horn knowledge bases such as QSQR and Magic-Sets. We begin with the following example:

Example A.1. This example is taken from [8]. Consider the following positive logic program P:

$$r_1: \quad ancestor(x, y) \leftarrow parent(x, y)$$
$$r_2: \quad ansestor(x, y) \leftarrow parent(x, z), ancestor(z, y)$$

where x, y, z are variables, *parent* is an extensional predicate with the meaning that $parent(x, y)$ is *true* if x is a parent of y, and *ancestor* is an intensional predicate with the meaning that $ancestor(x, y)$ is *true* if x is an ancestor of y.

The rule (clause) r_1 says that "*if x is a parent of y then x is an ancestor of y*", and the rule r_2 means "*if x is a parent of z and z is an ancestor of y then x is an ancestor of y*".

Let the query be $ancestor(john, x)$?, asking "*John is an ancestor of whom?*". The task is to find all the descendants of John. ∎

Evaluation of a query (e.g., in Example A.1) can be performed in two different ways, which have both advantages and disadvantages. The *bottom-up* strategy starts from the existing facts and infers new facts. This strategy always terminates and allows us to use set-at-a-time operations, which may be made efficient for accessing to the secondary storage. However, the bottom-up strategy is not goal-oriented, it can involve a lot of irrelevant computations. The *top-down* strategy starts from the query as a goal and uses rules from head to body to create more goals (i.e., subgoals). It is goal-oriented, but the computations are performed tuple-at-a-time so that the reduction of a goal to subgoals involves only a small amount of data, and the evaluation may not be terminated.

For evaluation of Datalog deductive databases (or Horn knowledge bases) queries, there are two types of *information passing*:

- *Unification*: in general, unification matches two terms t_1 and t_2 by finding a substitution of variables mapping M such that if M is applied to t_1 and M is applied to t_2 then the results are equal.
- *Sideways Information Passing Strategy* (SIPS) [7, 8]: given bindings for some variables of a predicate, we can solve the predicate with these bindings and thus obtain bindings for some other variables. These new bindings can be "passed" to another predicate in the same rule to restrict the computation for that predicate.

An *adornment* for an m-ary predicate p is a string "α" of length m made up of b *(bound)* and f *(free)*, denoted by p^α. By applying the SIPS for a program, we can generate adorned rules by using predicates with some arguments bound to constants, and the other arguments free. The general algorithm for adorning a rule is as follows [1]:

- all occurrences of each bound variable in the rule head are bound,
- all occurrences of constants are bound,
- if a variable x occurs in the rule body, then all occurrences of x in subsequent literals are bound,
- the otherwise is free.

A different ordering of the rule body would yield different adornments. We denote the adorned version of a program P by P^{ad}. See [1, 8] for more details.

Example A.2. The following program is the adorned version corresponding to the positive logic program P given in Example A.1 for the query $ancestor(john, x)$?:

$$\begin{aligned} r_1 : \quad & ancestor^{bf}(x, y) \leftarrow parent(x, y) \\ r_2 : \quad & ancestor^{bf}(x, y) \leftarrow parent(x, z), ancestor^{bf}(z, y) \\ & \text{Query:} \quad \leftarrow ancestor^{bf}(john, x)?. \end{aligned}$$

We denote this adorned program by P^{ad}. ∎

Restricting to query evaluation for Datalog deductive databases or Horn knowledge bases, there are top-down methods such as QSQ [1, 69, 71], QSQR [39], QoSaQ [71] and bottom-up methods such as Naive, (improved) Semi-naive evaluation and Magic-Sets [1, 7, 8]. We now present some basic definitions of the well-known evaluation methods such as QSQR and Magic-Sets. The QSQ approach (including QSQR, QoSaQ) is based on SLD-resolution, the magic-sets technique simulates QSQ. All of the methods based on QSQ (including bottom-up methods based on magic-sets transformation) are goal-directed.

A.1 Query-Subquery Recursive

The Query-Subquery (QSQ) method [69] is a top-down evaluation method based on backward chaining. As an advantage, it tries to access only relevant facts to answer the query.

The important key of this method is *subquery*. A goal, together with the program, determines a query. Similarly, a subgoal, together with the program, defines a subquery. In order to answer a query, each goal is expanded in a list of subgoals, which are then expanded in their turn.

The QSQ method for evaluating Datalog queries is based on SLD-resolution and executes operations in a set-oriented way. It uses the constants in the original query and "pushes" constants from goals to subgoals in the same way as pushing selections into joins. It uses "SIPS" to pass constants binding information from one literal to the next in the body of a rule. During the process of QSQ query evaluation, relation instances are stored in supplementary relations (denoted by $sup_0, sup_1, \ldots, sup_n$). Typically, these instances repeatedly acquire new tuples as the algorithm runs.

Query-Subquery Recursive (QSQR) is an algorithm based on the QSQ framework. The first version of QSQR evaluation method was formulated by Vieille in [69] for Datalog deductive databases. It is set-oriented and uses a tabulation technique. As discussed in [39], that version is incomplete. The work [39] corrects and generalizes the QSQR method for Horn knowledge bases to give a set-oriented depth-first search evaluation method. The correction depends on clearing global "input" relations for each iteration of the main loop. In their generalized version, they used substitutions instead of adornments and annotations (but has the effects of the annotated version). To deal with function symbols, they used a term-depth bound for atoms and substitutions occurring in the computation. They formulated two algorithms:

- Algorithm 1 is a tuple-at-a-time method, it is a combination of depth-first search and tabulation. In order to obtain all answers for a query, all the choices are systematically tried, and the process is repeated until no changes were made to the global answer variables (*i.e.*, *ans_*) during the last iteration of the main loop. The global input variables (*i.e.*, *input_*) are reset to empty relations for each iteration of the main loop.
- Algorithm 2 is a reformulation of Algorithm 1 using set-at-a-time technique. The reformulation is based on processing a set of goal atoms of the same predicate instead of processing a single goal atom. It is a mixture of depth-first search, breadth-first search and tabulation. By doing set-at-a-time, it reduces the number of accesses to the secondary storage. In order to avoid keeping unnecessary information it also uses the same variable for the whole sequence of supplements (*i.e.*, sup_i).

As stated in [39], the QSQR method has some disadvantages as its approach is like iterative deepening search. It allows redundant recomputations [39, Remark 3.2]:

> *"If we change Algorithm 1 by moving the call* `clear-input-var` *from the inside of the "repeat" loop to the place before the loop then it becomes incomplete. This was illustrated in [39, Example 3.1] and can be checked by using the implementation. Without clearing the global input_ relations for subsequent iterations of the main loop there are situations when ans_ atoms derived in some earlier steps cannot be exploited for the currently considered subquery to derive further results because the subquery is subsumed by a previously considered subquery and is then omitted. In other words, since the QSQR evaluation procedure is specified as a*

recursive function, newly derived ans_ atoms are not directly propagated to all recursive calls. That is, the intermediary ans_ relations are somehow local to each recursive call although the ans_ variables are global. This leads to the need to clear the input_ relations occasionally (e.g., at the beginning of each iteration of the main loop as in Algorithm 1, or after/before each recursive call) in order to allow recomputations using updated ans_ relations. Sometimes such recomputations are redundant. As observed by Vieille [71], the QSQR evaluation method is like iterative deepening search. It has both advantages and disadvantages."

A.2 Magic-Sets Transformation

The magic-sets technique for Datalog queries is a rule-rewriting method that generates from a given set of rules (clauses) a new set of rules, which is equivalent to the original set with respect to the original query. After rewriting, the new program (denoted by P^{mg}) can be evaluated by a simple bottom-up algorithm, usually by the improved semi-naive evaluation method. This approach takes the advantage of reducing irrelevant facts and restricting the search space. Thus, it combines the advantages of top-down and bottom-up methods.

We use the adorned program P^{ad} given in Example A.2 to demonstrate the magic-sets technique. From the original rules of P^{ad}, the magic-sets technique generates a new set of rules by the following steps (see [7, 8] for more details of each step):

1. Creating a new predicate $magic\_p^{\alpha}$ for each p^{α} in P^{ad}, the arity of the new predicate is the number of occurrences of b in the adornment α, and its arguments correspond to the bound arguments of p^{α}.

   For example, the following magic predicates are created for the adorned program P^{ad}:

   $magic\_ancestor^{bf}(x)$ and $magic\_ancestor^{bf}(z)$.

2. For each rule r in P^{ad}, and for each occurrence of an adorned predicate p^{α} in its body, generating a *magic rule* defining $magic\_p^{\alpha}$.

   For example, after generating a magic rule defining $magic\_ancestor^{bf}(z)$ from rule r_2 and the second body literal, we have:

   $magic\_ancestor^{bf}(z) \leftarrow magic\_ancestor^{bf}(x), parent(x, z).$

3. Modifying each rule in P^{ad} by adding an appropriate atom of the corresponding magic predicate to its body.

   For example, after modifying rule r_1, we have:

   $ancestor^{bf}(x, y) \leftarrow magic\_ancestor^{bf}(x), parent(x, y)$

   and modifying rule r_2, we have:

   $ancestor^{bf}(x, y) \leftarrow magic\_ancestor^{bf}(x), parent(x, z), ancestor^{bf}(z, y).$

4. Creating the *seed* for the query using the corresponding magic-sets predicate.

   For example, creating the seed from the query results in:

$$magic\_ancestor^{bf}(john).$$

Example A.3. The magic-sets rule-rewriting program corresponding to the adorned program P^{ad} given in Example A.2 is specified as follows:

$$r_1: \quad magic\_ancestor^{bf}(z) \leftarrow magic\_ancestor^{bf}(x), parent(x,z)$$
$$r_2: \quad ancestor^{bf}(x,y) \leftarrow magic\_ancestor^{bf}(x), parent(x,y)$$
$$r_3: \quad ancestor^{bf}(x,y) \leftarrow magic\_ancestor^{bf}(x), parent(x,z), ancestor^{bf}(z,y)$$
$$r_4: \quad magic\_ancestor^{bf}(john).$$

We denote this rule-rewriting program by P^{mg}. ■

There are close connections between magic-sets technique and QSQ approach. The predicate $magic\_p^{\alpha}$ (resp. p^{α}) in the magic-sets technique plays the role of the predicate $input\_p^{\alpha}$ (resp. ans_p^{α}) in the QSQ approach.

The Generalized Supplementary Magic Sets algorithm proposed by Beeri and Ramakrishnan [8] uses some special predicates called "supplementary magic predicates" in order to eliminate the duplicate work during the processing. For example, consider the magic-sets rule-rewriting program P^{mg} presented in Example A.3. The join of $magic\_ancestor^{bf}$ and $parent$ in the first magic rule r_1 is evaluated again in the third magic rule r_3. In order to reduce such a duplicate work, they store these results in special predicates called supplementary magic predicates. We refer the reader to [8] for details of this algorithm.

Example A.4. We give below the rewritten set of optimized rules for the program P^{mg} given in Example A.3 using the generalized supplementary magic-sets algorithm [8]:

$$supmagic_2^2(x,z) \leftarrow magic\_ancestor^{bf}(x), parent(x,z)$$
$$ancestor^{bf}(x,y) \leftarrow magic\_ancestor^{bf}(x), parent(x,y)$$
$$ancestor^{bf}(x,y) \leftarrow supmagic_2^2(x,z), ancestor^{bf}(z,y)$$
$$magic\_ancestor^{bf}(z) \leftarrow supmagic_2^2(x,z)$$
$$magic\_ancestor^{bf}(john).$$

■

After performing the magic-sets transformation using the generalized supplementary algorithm as in Example A.4, the obtained program can be evaluated by a bottom-up method such as the improved semi-naive evaluation method. In this case, the improved semi-naive evaluation method constructs a list $[R_1], \ldots, [R_n]$ of equivalence classes of intensional predicates with respect to their dependency [1]. Then, it computes the instances (relations) of the predicates in $[R_i]$ for each $1 \leq i \leq n$ in the increasing order, treating all the predicates in $[R_j]$ with $j < i$ as extensional predicates.

Both QSQR and Magic-Sets are the most well-known methods for evaluating queries to Datalog deductive databases or Horn knowledge bases. They are goal-directed. However, they have some disadvantages:

- the QSQR approach uses iterative deepening search and it allows redundant recomputations (e.g., see [39, Remark 3.2]),
- the Magic-Sets method applies breadth-first search and it is not always efficient (e.g., see Example 1.1).

It is worth developing other methods for evaluating queries to Horn knowledge bases, which are more efficient than QSQR and more adjustable than Magic-Sets.

Appendix B

Proof of Lemma 4.3 for the Case $T(r) = \mathit{false}$

We present here the proof of Lemma 4.3 for the case $T(r) = \mathit{false}$. The proof is very similar to the one for QSQN given by Nguyen in [45] and our revision [12], except for the case when the predicate p of $B_{i,j}$ is an intensional predicate. In this case, $T(p)$ can be either *true* or *false*. We present the full proof for this case here to make the text self-contained. We assume that the sets of fresh variables used for renaming variables of input program clauses in SLD-refutations and in Algorithm 2 are disjoint.

Proof. Suppose that $T(p) = \mathit{false}$. Recall that we prove the lemma by induction on the length of the mentioned SLD-refutation. Let $\theta_1, \dots, \theta_y$ be the sequence of mgu's used in the refutation. We have that $r(\bar{s})\theta_1 \dots \theta_y = r(\bar{s})\theta$. Suppose that the first step of the refutation of $P \cup I \cup \{\leftarrow r(\bar{s})\}$ uses an input program clause $\varphi'_i = (A'_i \leftarrow B'_{i,1}, \dots, B'_{i,n_i})$, which is a variant of a program clause $\varphi_i = (A_i \leftarrow B_{i,1}, \dots, B_{i,n_i})$ of P, resulting in the resolvent $\leftarrow (B'_{i,1}, \dots, B'_{i,n_i})\theta_1$. Let $k_1 = 2$, $k_{n_i+1} = y + 1$ and suppose that, for $1 \leq j \leq n_i$,

$$\text{the fragment for processing } \leftarrow B'_{i,j}\theta_1 \dots \theta_{k_j-1} \text{ of the refutation of } P \cup I \cup \{\leftarrow r(\bar{s})\} \text{ uses mgu's } \theta_{k_j}, \dots, \theta_{k_{j+1}-1}. \tag{B.1}$$

Thus, after processing the atom $B'_{i,j-1}$ for $2 \leq j \leq n_i+1$, the next goal of the refutation of $\leftarrow r(\bar{s})$ is $\leftarrow (B'_{i,j}, \dots, B'_{i,n_i})\theta_1 \dots \theta_{k_j-1}$. (If $j = n_i + 1$ then the goal is empty.)

Let ϱ be a renaming substitution such that $\varphi'_i = \varphi_i\varrho$. Thus, $B'_{i,j} = B_{i,j}\varrho$ for $1 \leq j \leq n_i$. We can assume that ϱ does not use any variable occurring in $\bar{s}$. Thus,

$$\bar{s} = \bar{s}\varrho. \tag{B.2}$$

Since $\theta_1 = mgu(r(\bar{s}), A'_i)$ and $A'_i = A_i\varrho$ and by (B.2), it follows that $r(\bar{s})\varrho\theta_1 = r(\bar{s})\theta_1 = A'_i\theta_1 = A_i\varrho\theta_1$ and hence $\varrho\theta_1$ is a unifier for $r(\bar{s})$ and A_i. Let γ_0 be an mgu Algorithm 2 used to unify $r(\bar{s})$ with A_i when processing $\bar{s}$ for the edge $(\mathit{input\_r}, \mathit{pre\_filter}_i)$. Thus, there exists a substitution η_0 such that $\gamma_0\eta_0 = \varrho\theta_1$.

Let $\bar{t}_0 = \bar{s}\gamma_0$ and $\delta_0 = (\gamma_0)_{|\mathit{post\_vars}(\mathit{pre\_filter}_i)}$.

Consider the base case, which occurs when $n_i = 0$ and the SLD-refutation has the length one. By (B.2) and the fact $\gamma_0\eta_0 = \varrho\theta_1$, we have that

$$\bar{s}\theta_1 = \bar{s}\varrho\theta_1 = \bar{s}\gamma_0\eta_0 = \bar{t}_0\eta_0. \tag{B.3}$$

Thus, $\bar{s}\theta_1$ is an instance of $\bar{t}_0$. Since $post\_vars(pre\_filter_i) = \emptyset$, the subquery $(\bar{t}_0, \varepsilon)$ was transferred through the edge $(pre\_filter_i, post\_filter_i)$. Hence, $tuples(ans\_r)$ contains $\bar{s}''$ such that $\bar{t}_0$ is an instance of a fresh variant of $\bar{s}''$. Since $\bar{s}\theta = \bar{s}\theta_1$, it follows that, $\bar{s}\theta$ is an instance of a variant of $\bar{s}''$.

Let us consider the induction step. We have that $n_i \geq 1$. We will refer to the data structures used by Algorithm 2. We first prove the following remark:

Remark B.1. *Let $1 \leq j \leq n_i$, $v = filter_{i,j}$, $u = filter_{i,j-1}$ if $j > 1$, and $u = pre\_filter_i$ otherwise. If $(\bar{t}_{j-1}, \delta_{j-1})$ is a subquery transferred through (u, v) at some step and there exists a substitution η such that*

$$(A_i, (B_{i,j}, \dots, B_{i,n_i}))\varrho\theta_1 \dots \theta_{k_j-1} = (r(\bar{t}_{j-1}), (B_{i,j}, \dots, B_{i,n_i})\delta_{j-1})\eta, \tag{B.4}$$

then there exist a subquery $(\bar{t}_j, \delta_j)$ transferred through $(v, succ(v))$ at some step and a substitution η' such that

$$(A_i, (B_{i,j+1}, \dots, B_{i,n_i}))\varrho\theta_1 \dots \theta_{k_{j+1}-1} = (r(\bar{t}_j), (B_{i,j+1}, \dots, B_{i,n_i})\delta_j)\eta'. \tag{B.5}$$

Suppose the premises of this remark hold. Without loss of generality we assume that:

$$\text{if } (kind(v) = extensional \text{ and } T(v) = true) \text{ or } kind(v) = intensional \text{ then the subquery } (\bar{t}_{j-1}, \delta_{j-1}) \text{ was added to } subqueries(v). \tag{B.6}$$

Since $B'_{i,j} = B_{i,j}\varrho$ and (B.4), we have that:

$$(\leftarrow B'_{i,j}\theta_1 \dots \theta_{k_j-1}) = (\leftarrow B_{i,j}\varrho\theta_1 \dots \theta_{k_j-1}) = (\leftarrow B_{i,j}\delta_{j-1}\eta). \tag{B.7}$$

Since the term-depth of $B_{i,j}\delta_{j-1}\eta = B'_{i,j}\theta_1 \dots \theta_{k_j-1}$ is not greater than l, the term-depth of $B_{i,j}\delta_{j-1}$ is also not greater than l. By (B.1), (B.7) and Lifting Lemma 2.2 we have that

$$\text{there exists a refutation of } P \cup I \cup \{\leftarrow B_{i,j}\delta_{j-1}\} \text{ using the leftmost selection function and mgu's } \theta'_{k_j}, \dots, \theta'_{k_{j+1}-1} \text{ such that the term-depths of goals are not greater than } l \text{ and } \eta\theta_{k_j} \dots \theta_{k_{j+1}-1} = \theta'_{k_j} \dots \theta'_{k_{j+1}-1}\mu \text{ for some substitution } \mu. \tag{B.8}$$

Consider the case when the predicate $p = pred(v)$ of $B_{i,j}$ is an extensional predicate. Thus,

$$k_{j+1} = k_j + 1 \tag{B.9}$$

and

$$B_{i,j}\delta_{j-1}\theta'_{k_j} = p(\bar{t}')\sigma\theta'_{k_j} \tag{B.10}$$

where $p(\bar{t}')\sigma$ is the input program clause used for resolving $\leftarrow B_{i,j}\delta_{j-1}$, with $\bar{t}' \in I(p)$ and σ being a renaming substitution. Regarding the transfer of the subquery $(\bar{t}_{j-1}, \delta_{j-1})$ through (u, v), under the assumption (B.6), Algorithm 2 unifies $atom(v)\delta_{j-1} = B_{i,j}\delta_{j-1}$ with a fresh variant $p(\bar{t}')\sigma'$ of $p(\bar{t}')$, where σ' is a renaming substitution, resulting in an mgu γ (by (B.10), $B_{i,j}\delta_{j-1}$ and $p(\bar{t}')\sigma'$ are unifiable) and then transfers the subquery $(\bar{t}_{j-1}\gamma, (\delta_{j-1}\gamma)_{|post\_vars(v)})$ through $(v, succ(v))$. Let

$$\bar{t}_j = \bar{t}_{j-1}\gamma \ \text{ and } \ \delta_j = (\delta_{j-1}\gamma)_{|post\_vars(v)}. \tag{B.11}$$

We have that $\sigma = \sigma'\sigma''$ for some renaming substitution σ'' such that

$$\sigma'' \text{ does not use variables of } \bar{t}_{j-1},\ \delta_{j-1} \text{ and } pre\_vars(v). \tag{B.12}$$

Thus $B_{i,j}\delta_{j-1}\sigma''\theta'_{k_j} = B_{i,j}\delta_{j-1}\theta'_{k_j}$, and by (B.10) and the fact $\sigma = \sigma'\sigma''$, we have that

$$(B_{i,j}\delta_{j-1})\sigma''\theta'_{k_j} = B_{i,j}\delta_{j-1}\theta'_{k_j} = p(\bar{t}')\sigma\theta'_{k_j} = (p(\bar{t}')\sigma')\sigma''\theta'_{k_j}.$$

Hence, $B_{i,j}\delta_{j-1}$ and $p(\bar{t}')\sigma'$ are unifiable using $\sigma''\theta'_{k_j}$, while γ is an mgu for them. Hence

$$\sigma''\theta'_{k_j} = \gamma\mu' \tag{B.13}$$

for some substitution μ'. Let $\eta' = \mu'\mu$. We have that:

$$\begin{aligned}
& (A_i, (B_{i,j+1}, \dots, B_{i,n_i}))\varrho\theta_1 \dots \theta_{k_{j+1}-1} \\
= \ & ((A_i, (B_{i,j+1}, \dots, B_{i,n_i}))\varrho\theta_1 \dots \theta_{k_j-1})\theta_{k_j} \dots \theta_{k_{j+1}-1} \\
= \ & (r(\bar{t}_{j-1}), (B_{i,j+1}, \dots, B_{i,n_i})\delta_{j-1})\eta\theta_{k_j} \dots \theta_{k_{j+1}-1} \ \ \text{(by the assumption (B.4))} \\
= \ & (r(\bar{t}_{j-1}), (B_{i,j+1}, \dots, B_{i,n_i})\delta_{j-1})\theta'_{k_j} \dots \theta'_{k_{j+1}-1}\mu \ \ \text{(by (B.8))} \\
= \ & (r(\bar{t}_{j-1}), (B_{i,j+1}, \dots, B_{i,n_i})\delta_{j-1})\sigma''\theta'_{k_j} \dots \theta'_{k_{j+1}-1}\mu \ \ \text{(by (B.12))} \\
= \ & (r(\bar{t}_{j-1}), (B_{i,j+1}, \dots, B_{i,n_i})\delta_{j-1})\gamma\mu'\mu \ \ \text{(by (B.9) and (B.13))} \\
= \ & (r(\bar{t}_j), (B_{i,j+1}, \dots, B_{i,n_i})\delta_j)\eta' \ \ \text{(by (B.11) and the fact } \eta' = \mu'\mu).
\end{aligned}$$

We have shown (B.5) and thus proved Remark B.1 for the case when the predicate of $B_{i,j}$ is extensional.

Now consider the case when the predicate p of $B_{i,j}$ is an intensional predicate.

By the assumption (B.6), the subquery $(\bar{t}_{j-1}, \delta_{j-1})$ was also added to $unprocessed\_subqueries_2(v)$. Let $B_{i,j}\delta_{j-1} = p(\bar{t}'_j)$. If $T(p) = true$ (resp. $T(p) = false$) then the pair $(\bar{t}'_j, \bar{t}'_j)$ (resp. tuple $\bar{t}'_j$) was transferred through the edge $(v, input\_p)$, hence there must exist some tuple pair $(\bar{t}, \bar{t}')$ (resp. tuple $\bar{t}'$) more general than a fresh variant of $(\bar{t}'_j, \bar{t}'_j)$ (resp. $\bar{t}'_j$) that was added to $tuple\_pairs(input\_p)$ (resp. $tuples(input\_p)$) at some step, and thus $(\bar{t}, \bar{t}')\lambda = (\bar{t}'_j, \bar{t}'_j)\lambda'$ (resp. $\bar{t}'\lambda = \bar{t}'_j\lambda'$) for some substitution λ that uses only variables from $\bar{t}$, $\bar{t}'$ (resp. $\bar{t}'$) and a renaming substitution λ' with domain contained in $Vars(\bar{t}'_j)$. Hence, $(\bar{t}, \bar{t}')\alpha = (\bar{t}'_j, \bar{t}'_j)$ (resp. $\bar{t}'\alpha = \bar{t}'_j$) for the substitution $\alpha = \lambda(\lambda')^{-1}$. We can assume that α uses only variables from $\bar{t}$, $\bar{t}'$ and $\bar{t}'_j$ (resp. $\bar{t}'$ and $\bar{t}'_j$). Thus,

$$B_{i,j}\delta_{j-1} = p(\bar{t}'_j) = p(\bar{t}')\alpha \quad \text{if } T(p) = \mathit{false}, \tag{B.14}$$

and

$$B_{i,j}\delta_{j-1} = p(\bar{t}'_j) = p(\bar{t})\alpha = p(\bar{t}')\alpha \quad \text{if } T(p) = \mathit{true}. \tag{B.15}$$

By (B.8) and Lifting Lemma 2.2, it follows that there exists a refutation of $P \cup I \cup \{\leftarrow p(\bar{t})\}$ if $T(p) = \mathit{true}$ (resp. $P \cup I \cup \{\leftarrow p(\bar{t}')\}$ if $T(p) = \mathit{false}$) using the leftmost selection function and mgu's $\theta''_{k_j}, \ldots, \theta''_{k_{j+1}-1}$ such that the term-depths of the goals are not greater than l and

$$\alpha\theta'_{k_j} \ldots \theta'_{k_{j+1}-1} = \theta''_{k_j} \ldots \theta''_{k_{j+1}-1}\beta \tag{B.16}$$

for some substitution β. By the inductive assumption, $\mathit{tuples}(\mathit{ans\_p})$ contains a tuple $\bar{t}''$ such that $\bar{t}'\theta''_{k_j} \ldots \theta''_{k_{j+1}-1}$ is an instance of a variant of $\bar{t}''$. Since

$$\begin{array}{rcll} B_{i,j}\delta_{j-1}\theta'_{k_j} \ldots \theta'_{k_{j+1}-1} & = & p(\bar{t}')\alpha\theta'_{k_j} \ldots \theta'_{k_{j+1}-1} & \text{(by (B.14) and (B.15))} \\ & = & p(\bar{t}')\theta''_{k_j} \ldots \theta''_{k_{j+1}-1}\beta & \text{(by (B.16)),} \end{array}$$

it follows that

$$B_{i,j}\delta_{j-1}\theta'_{k_j} \ldots \theta'_{k_{j+1}-1} \text{ is an instance of a variant of } p(\bar{t}''). \tag{B.17}$$

From a certain moment there were both $(\bar{t}_{j-1}, \delta_{j-1}) \in \mathit{subqueries}(v)$ and $\bar{t}'' \in \mathit{tuples}(\mathit{ans\_p})$. Hence, at some step Algorithm 2 unified $\mathit{atom}(v)(\delta_{j-1}) = B_{i,j}\delta_{j-1}$ with a fresh variant $p(\bar{t}'')\sigma$ of $p(\bar{t}'')$, where σ is a renaming substitution. The atom $p(\bar{t}'')\sigma$ does not contain variables of $\bar{t}_{j-1}$, δ_{j-1}, $\mathit{pre\_vars}(v)$ and $\theta'_{k_j} \ldots \theta'_{k_{j+1}-1}$. By (B.17), $B_{i,j}\delta_{j-1}$ and $p(\bar{t}'')\sigma$ are unifiable. Let the resulting mgu be γ and let

$$\bar{t}_j = \bar{t}_{j-1}\gamma \text{ and } \delta_j = (\delta_{j-1}\gamma)_{|\mathit{post\_vars}(v)}. \tag{B.18}$$

Algorithm 2 then transferred the subquery $(\bar{t}_j, \delta_j)$ through $(v, \mathit{succ}(v))$.

By (B.17), $B_{i,j}\delta_{j-1}\theta'_{k_j} \ldots \theta'_{k_{j+1}-1}$ is an instance of $p(\bar{t}'')\sigma$. Let ρ be a substitution with domain contained in $\mathit{Vars}(p(\bar{t}'')\sigma)$ such that $B_{i,j}\delta_{j-1}\theta'_{k_j} \ldots \theta'_{k_{j+1}-1} = p(\bar{t}'')\sigma\rho$. We have that

$$\text{the domain of } \rho \text{ does not contain variables of } \bar{t}_{j-1}, \delta_{j-1}, \mathit{pre\_vars}(v) \text{ and } \theta'_{k_j} \ldots \theta'_{k_{j+1}-1} \tag{B.19}$$

and $\theta'_{k_j} \ldots \theta'_{k_{j+1}-1} \cup \rho$ is a unifier for $B_{i,j}\delta_{j-1}$ and $p(\bar{t}'')\sigma$. As γ is an mgu for $B_{i,j}\delta_{j-1}$ and $p(\bar{t}'')\sigma$, we have that

$$\gamma\mu' = (\theta'_{k_j} \ldots \theta'_{k_{j+1}-1} \cup \rho) \tag{B.20}$$

for some substitution μ'. Let $\eta' = \mu'\mu$. We have that:

$$\begin{array}{rl} & (A_i, (B_{i,j+1}, \ldots, B_{i,n_i}))\varrho\theta_1 \ldots \theta_{k_{j+1}-1} \\ = & (r(\bar{t}_{j-1}), (B_{i,j+1}, \ldots, B_{i,n_i})\delta_{j-1})\theta'_{k_j} \ldots \theta'_{k_{j+1}-1}\mu \ \text{ (as shown before)} \\ = & (r(\bar{t}_{j-1}), (B_{i,j+1}, \ldots, B_{i,n_i})\delta_{j-1})(\theta'_{k_j} \ldots \theta'_{k_{j+1}-1} \cup \rho)\mu \ \text{ (by (B.19))} \\ = & (r(\bar{t}_{j-1}), (B_{i,j+1}, \ldots, B_{i,n_i})\delta_{j-1})\gamma\mu'\mu \ \text{ (by (B.20))} \\ = & (r(\bar{t}_j), (B_{i,j+1}, \ldots, B_{i,n_i})\delta_j)\eta' \ \text{ (by (B.18) and the fact } \eta' = \mu'\mu). \end{array}$$

We have shown (B.5) and thus proved Remark B.1 for the case when the predicate of $B_{i,j}$ is intensional. This completes the proof of this remark. ∎

Recall that $\bar{t}_0 = \bar{s}\gamma_0$ and $\delta_0 = (\gamma_0)_{|post\_vars(pre\_filter_i)}$ and $k_1 = 2$. The subquery $(\bar{t}_0, \delta_0)$ was transferred through the edge $(pre\_filter_i, filter_{i,1})$. Observe that the premises of Remark B.1 hold for $j = 1$ and for the subquery $(\bar{t}_0, \delta_0)$ using $\eta = \eta_0$. Hence there exist a subquery $(\bar{t}_1, \delta_1)$ transferred through $(filter_{i,1}, succ(filter_{i,1}))$ at some step and a substitution η_1 such that

$$(A_i, (B_{i,2}, \dots, B_{i,n_i}))\varrho\theta_1 \dots \theta_{k_2-1} = (r(\bar{t}_1), (B_{i,2}, \dots, B_{i,n_i})\delta_1)\eta_1.$$

For each $1 < j \le n_i$, we can apply Remark B.1 to obtain a subquery $(\bar{t}_j, \delta_j)$ and η_j (for η'). Since $post\_vars(filter_{i,n_i}) = \emptyset$, it follows that, for $j = n_i$, we have that $(\bar{t}_{n_i}, \varepsilon)$ is a subquery transferred through $(filter_{i,n_i}, post\_filter_i)$ at some step and

$$A_i\varrho\theta_1 \dots \theta_{k_{n_i+1}-1} = r(\bar{t}_{n_i})\eta_{n_i}.$$

Since $k_{n_i+1} = y + 1$ and $\theta = (\theta_1 \dots \theta_y)_{|Vars(\bar{s})}$, it follows that

$$r(\bar{s})\theta = r(\bar{s})\theta_1 \dots \theta_y = A'_i\theta_1 \dots \theta_y = A_i\varrho\theta_1 \dots \theta_y = r(\bar{t}_{n_i})\eta_{n_i}.$$

Thus, $\bar{s}\theta$ is an instance of $\bar{t}_{n_i}$. Since $(\bar{t}_{n_i}, \varepsilon)$ was transferred through the edge $(filter_{i,n_i}, post\_filter_i)$, $tuples(ans\_r)$ contains $\bar{s}''$ such that $\bar{t}_{n_i}$ is an instance of a fresh variant of $\bar{s}''$. It follows that, $\bar{s}\theta$ is an instance of a variant of $\bar{s}''$. This completes the proof of Lemma 4.3 for the case $T(r) = false$. ∎

Appendix C

Functions and Procedures Used for Algorithm 2

In this section, we present a list of all functions and procedures that are related to the processing of Algorithm 2 (on page 38) such as `fire2` and `transfer2`. They are modified versions of procedures used for Algorithm 1. The related procedures used for `fire2` and `transfer2` such as `add-tuple-pair` and `compute-gamma` are also listed.

Procedure add-tuple-pair($\bar{t}, \bar{t}', \Gamma$)

Purpose: add the pair of tuples $(\bar{t}, \bar{t}')$ to Γ, but keep in Γ only the most general pairs.

let $(\bar{t}_2, \bar{t}_2')$ be a fresh variant of $(\bar{t}, \bar{t}')$;
if $(\bar{t}_2, \bar{t}_2')$ *is not an instance of any pair from* Γ **then**
  delete from Γ all pairs that are instances of $(\bar{t}_2, \bar{t}_2')$;
  add $(\bar{t}_2, \bar{t}_2')$ to Γ

Procedure compute-gamma

Purpose: a macro used in procedure `fire2`

if $T(p) = false$ **then**
  foreach $(\bar{t}, \delta) \in unprocessed\_subqueries_2(u)$ **do**
    let $p(\bar{t}') = atom(u)\delta$;
    `add-tuple`($\bar{t}', \Gamma$)
else if $(j < n_i)$ *or* (p *is not the predicate of* A_i) **then**
  foreach $(\bar{t}, \delta) \in unprocessed\_subqueries_2(u)$ **do**
    let $p(\bar{t}') = atom(u)\delta$;
    `add-tuple-pair`($\bar{t}', \bar{t}', \Gamma$)
else
  foreach $(\bar{t}, \delta) \in unprocessed\_subqueries_2(u)$ **do**
    let $p(\bar{t}') = atom(u)\delta$;
    `add-tuple-pair`($\bar{t}', \bar{t}, \Gamma$)

Procedure fire2(u, v)

Global data: a Horn knowledge base (P, I), a QSQN-TRE $N = (V, E, T, C)$ of P, and a term-depth bound l.

Input: an edge $(u, v) \in E$ such that `active-edge`(u, v) holds.

```
if u is input_p or ans_p then
    transfer2(unprocessed(u, v), u, v);
    unprocessed(u, v) := ∅
else if u is filter_{i,j} and kind(u) = extensional and T(u) = true then
    let p = pred(u) and set Γ := ∅;
    foreach (t̄, δ) ∈ unprocessed_subqueries(u) do
        foreach t̄' ∈ I(p) do
            if atom(u)δ is unifiable with a fresh variant of p(t̄') by an mgu γ then
                add-subquery(t̄γ, (δγ)|post_vars(u), Γ, v)
    unprocessed_subqueries(u) := ∅;
    transfer2(Γ, u, v)
else if u is filter_{i,j} and kind(u) = intensional then
    let p = pred(u) and set Γ := ∅;
    if v = input_p then
        compute-gamma;
        unprocessed_subqueries_2(u) := ∅;
    else
        foreach (t̄, δ) ∈ unprocessed_subqueries(u) do
            foreach t̄' ∈ tuples(ans_p) do
                if atom(u)δ is unifiable with a fresh variant of p(t̄') by an mgu γ
                then
                    add-subquery(t̄γ, (δγ)|post_vars(u), Γ, v)
        unprocessed_subqueries(u) := ∅;
        if unprocessed_tuples(u) ≠ ∅ then
            foreach t̄ ∈ unprocessed_tuples(u) do
                foreach (t̄', δ) ∈ subqueries(u) do
                    if atom(u)δ is unifiable with a fresh variant of p(t̄) by an mgu
                    γ then
                        add-subquery(t̄'γ, (δγ)|post_vars(u), Γ, v)
            unprocessed_tuples(u) := ∅
    transfer2(Γ, u, v)
```

Procedure transfer2(D, u, v) 1/2

Global data: a Horn knowledge base (P, I), a QSQN-TRE $N = (V, E, T, C)$ of P, and a term-depth bound l.
Input: data D to transfer through the edge $(u, v) \in E$.

if $D = \emptyset$ **then return**;
if *u is input_p and* $T(p) = true$ **then**
  $\Gamma := \emptyset$;
  foreach $(\bar{t}, \bar{t}') \in D$ **do**
    if *$p(\bar{t})$ and $atom(v)$ are unifiable by an mgu γ* **then**
      `add-subquery`$(\bar{t}'\gamma, \gamma_{|post\_vars(v)}, \Gamma, succ(v))$
  `transfer2`$(\Gamma, v, succ(v))$
else if *v is input_p and* $T(p) = true$ **then**
  foreach $(\bar{t}, \bar{t}') \in D$ **do**
    let $(\bar{t}_2, \bar{t}_2')$ be a fresh variant of $(\bar{t}, \bar{t}')$;
    if *$(\bar{t}_2, \bar{t}_2')$ is not an instance of any pair from $tuple\_pairs(v)$* **then**
      foreach $(\bar{t}_3, \bar{t}_3') \in tuple\_pairs(v)$ **do**
        if *$(\bar{t}_3, \bar{t}_3')$ is an instance of $(\bar{t}_2, \bar{t}_2')$* **then**
          delete $(\bar{t}_3, \bar{t}_3')$ from $tuple\_pairs(v)$;
          foreach $(v, w) \in E$ **do** delete $(\bar{t}_3, \bar{t}_3')$ from $unprocessed(v, w)$;
      add $(\bar{t}_2, \bar{t}_2')$ to $tuple\_pairs(v)$;
      foreach $(v, w) \in E$ **do** add $(\bar{t}_2, \bar{t}_2')$ to $unprocessed(v, w)$;
else if *v is $filter_{i,n_i}$, $kind(v) = intensional$, $pred(v) = p$ and $T(p) = true$* **then**
  foreach $(\bar{t}, \delta) \in D$ **do**
    if `term-depth`$(atom(v)\delta) \leq l$ **then**
      if *no subquery in $subqueries(v)$ is more general than $(\bar{t}, \delta)$* **then**
        delete from $subqueries(v)$ all subqueries less general than $(\bar{t}, \delta)$;
        delete from $unprocessed\_subqueries_2(v)$ all subqueries less general than $(\bar{t}, \delta)$;
        add $(\bar{t}, \delta)$ to both $subqueries(v)$ and $unprocessed\_subqueries_2(v)$
else if *u is input_p* **then**
  $\Gamma := \emptyset$;
  foreach $\bar{t} \in D$ **do**
    if *$p(\bar{t})$ and $atom(v)$ are unifiable by an mgu γ* **then**
      `add-subquery`$(\bar{t}\gamma, \gamma_{|post\_vars(v)}, \Gamma, succ(v))$
  `transfer2`$(\Gamma, v, succ(v))$
else if *u is ans_p* **then** $unprocessed\_tuples(v) := unprocessed\_tuples(v) \cup D$;

Procedure transfer2(D, u, v) *(continued)* 2/2

```
else if v is input_p or ans_p then
    foreach t̄ ∈ D do
        let t̄′ be a fresh variant of t̄;
        if t̄′ is not an instance of any tuple from tuples(v) then
            foreach t̄″ ∈ tuples(v) do
                if t̄″ is an instance of t̄′ then
                    delete t̄″ from tuples(v);
                    foreach (v, w) ∈ E do delete t̄″ from unprocessed(v, w);
            if v is input_p then
                add t̄′ to tuples(v);
                foreach (v, w) ∈ E do add t̄′ to unprocessed(v, w);
            else
                add t̄ to tuples(v);
                foreach (v, w) ∈ E do add t̄ to unprocessed(v, w);
else if v is filter_{i,j} and kind(v) = extensional and T(v) = false then
    let p = pred(v) and set Γ := ∅;
    foreach (t̄, δ) ∈ D do
        if term-depth(atom(v)δ) ≤ l then
            foreach t̄′ ∈ I(p) do
                if atom(v)δ is unifiable with a fresh variant of p(t̄′) by an mgu γ
                then
                    add-subquery(t̄γ, (δγ)|post_vars(v), Γ, succ(v))
    transfer2(Γ, v, succ(v))
else if v is filter_{i,j} and (kind(v) = extensional and T(v) = true or
kind(v) = intensional) then
    foreach (t̄, δ) ∈ D do
        if term-depth(atom(v)δ) ≤ l then
            if no subquery in subqueries(v) is more general than (t̄, δ) then
                delete from subqueries(v) all subqueries less general than (t̄, δ);
                delete from unprocessed_subqueries(v) all subqueries less general
                than (t̄, δ);
                add (t̄, δ) to both subqueries(v) and unprocessed_subqueries(v);
                if kind(v) = intensional then
                    delete from unprocessed_subqueries_2(v) all subqueries less
                    general than (t̄, δ);
                    add (t̄, δ) to unprocessed_subqueries_2(v)
else // v is of the form post_filter_i
    Γ := {t̄ | (t̄, ε) ∈ D};
    transfer2(Γ, v, succ(v))
```

Appendix D

Functions and Procedures Used for Algorithm 3

In this section, we present a list of all functions and procedures that are used for Algorithm 3 (on page 57). This algorithm uses the function `active-edge`(u, v) (on page 28) and the procedure `fire3`(u, v) (on page 116). The related procedures such as `add-subquery3`, `add-ta-pair`, `compute-gamma3` and `transfer3` are also listed. They are modified versions of procedures used for Algorithm 2.

Procedure add-subquery3$(q(\bar{t}), \delta, \Gamma, v)$

Purpose: add the subquery $(q(\bar{t}), \delta)$ to Γ, but keep in Γ only the most general subqueries w.r.t. v.

if `term-depth`$(\bar{t}) \leq l$ *and* `term-depth`$(\delta) \leq l$ *and no subquery in* Γ *is more general than* $(q(\bar{t}), \delta)$ *w.r.t.* v **then**
  delete from Γ all subqueries less general than $(q(\bar{t}), \delta)$ w.r.t. v;
  add $(q(\bar{t}), \delta)$ to Γ

Procedure compute-gamma3

Purpose: a macro used in procedure `fire3`

if $T(p) = false$ **then**
  foreach $(q(\bar{t}), \delta) \in unprocessed\_subqueries_2(u)$ **do**
    let $p(\bar{t}') = atom(u)\delta$;
    `add-tuple`$(\bar{t}', \Gamma)$
else if $(j < n_i)$ **then**
  foreach $(q(\bar{t}), \delta) \in unprocessed\_subqueries_2(u)$ **do**
    let $p(\bar{t}') = atom(u)\delta$;
    `add-ta-pair`$(\bar{t}', p(\bar{t}'), \Gamma)$
else
  foreach $(q(\bar{t}), \delta) \in unprocessed\_subqueries_2(u)$ **do**
    let $p(\bar{t}') = atom(u)\delta$;
    `add-ta-pair`$(\bar{t}', q(\bar{t}), \Gamma)$

Procedure add-ta-pair($\bar{t}, q(\bar{t}'), \Gamma$)

Purpose: add the (tuple-atom) pair $(\bar{t}, q(\bar{t}'))$ to Γ, but keep in Γ only the most general pairs.

let $(\bar{t}_2, \bar{t}'_2)$ be a fresh variant of $(\bar{t}, \bar{t}')$;
if $(\bar{t}_2, q(\bar{t}'_2))$ *is not an instance of any pair from* Γ **then**
    delete from Γ all pairs that are instances of $(\bar{t}_2, q(\bar{t}'_2))$;
    add $(\bar{t}_2, q(\bar{t}'_2))$ to Γ

Procedure fire3(u, v)

Global data: a Horn knowledge base (P, I), a QSQN-rTRE $N = (V, E, T, C)$ of P, and a term-depth bound l.
Input: an edge $(u, v) \in E$ such that `active-edge`(u, v) holds.

if u *is input_p or ans_p* **then**
    `transfer3`(*unprocessed*$(u, v), u, v$);
    unprocessed$(u, v) := \emptyset$
else if u *is filter*$_{i,j}$ *and kind*$(u) =$ *extensional and* $T(u) =$ *true* **then**
    let $p =$ *pred*(u) and set $\Gamma := \emptyset$;
    foreach $(q(\bar{t}), \delta) \in$ *unprocessed_subqueries*(u) **do**
        foreach $\bar{t}' \in I(p)$ **do**
            if *atom*$(u)\delta$ *is unifiable with a fresh variant of* $p(\bar{t}')$ *by an mgu* γ **then**
                `add-subquery3`$(q(\bar{t})\gamma, (\delta\gamma)_{|post\_vars(u)}, \Gamma, v)$
    unprocessed_subqueries$(u) := \emptyset$;
    `transfer3`(Γ, u, v)
else if u *is filter*$_{i,j}$ *and kind*$(u) =$ *intensional* **then**
    let $p =$ *pred*(u) and set $\Gamma := \emptyset$;
    if $v =$ *input_p* **then**
        `compute-gamma3`;
        unprocessed_subqueries$_2(u) := \emptyset$;
    else
        foreach $(q(\bar{t}), \delta) \in$ *unprocessed_subqueries*(u) **do**
            foreach $\bar{t}' \in$ *tuples*$($*ans_p*$)$ **do**
                if *atom*$(u)\delta$ *is unifiable with a fresh variant of* $p(\bar{t}')$ *by an mgu* γ **then**
                    `add-subquery3`$(q(\bar{t})\gamma, (\delta\gamma)_{|post\_vars(u)}, \Gamma, v)$
        unprocessed_subqueries$(u) := \emptyset$;
        if *unprocessed_tuples*$(u) \neq \emptyset$ **then**
            foreach $\bar{t} \in$ *unprocessed_tuples*(u) **do**
                foreach $(q(\bar{t}'), \delta) \in$ *subqueries*(u) **do**
                    if *atom*$(u)\delta$ *is unifiable with a fresh variant of* $p(\bar{t})$ *by an mgu* γ **then**
                        `add-subquery3`$(q(\bar{t}')\gamma, (\delta\gamma)_{|post\_vars(u)}, \Gamma, v)$
            unprocessed_tuples$(u) := \emptyset$
    `transfer3`(Γ, u, v)

Procedure transfer3(D, u, v) 1/2

Global data: a Horn knowledge base (P, I), a QSQN-rTRE $N = (V, E, T, C)$ of P, and a term-depth bound l.

Input: data D to transfer through the edge $(u, v) \in E$.

if $D = \emptyset$ **then return**;
if u *is input_p and* $T(p) = true$ **then**
  $\Gamma := \emptyset$;
  foreach $(\bar{t}, q(\bar{t}')) \in D$ **do**
    if $p(\bar{t})$ *and atom*(v) *are unifiable by an mgu* γ **then**
      `add-subquery3`$(q(\bar{t}')\gamma, \gamma_{|post\_vars(v)}, \Gamma, succ(v))$
  `transfer3`$(\Gamma, v, succ(v))$
else if v *is input_p and* $T(p) = true$ **then**
  foreach $(\bar{t}, q(\bar{t}')) \in D$ **do**
    let $(\bar{t}_2, \bar{t}'_2)$ be a fresh variant of $(\bar{t}, \bar{t}')$;
    if $(\bar{t}_2, q(\bar{t}'_2))$ *is not an instance of any pair from ta_pairs*(v) **then**
      foreach $(\bar{t}_3, q(\bar{t}'_3)) \in ta\_pairs(v)$ **do**
        if $(\bar{t}_3, q(\bar{t}'_3))$ *is an instance of* $(\bar{t}_2, q(\bar{t}'_2))$ **then**
          delete $(\bar{t}_3, q(\bar{t}'_3))$ from $ta\_pairs(v)$;
          foreach $(v, w) \in E$ **do**
            delete $(\bar{t}_3, q(\bar{t}'_3))$ from $unprocessed(v, w)$
      add $(\bar{t}_2, q(\bar{t}'_2))$ to $ta\_pairs(v)$;
      foreach $(v, w) \in E$ **do** add $(\bar{t}_2, q(\bar{t}'_2))$ to $unprocessed(v, w)$;
else if v *is filter*$_{i,n_i}$, $kind(v) = intensional$, $pred(v) = p$ *and* $T(p) = true$ **then**
  foreach $(q(\bar{t}), \delta) \in D$ **do**
    if `term-depth`$(atom(v)\delta) \leq l$ **then**
      if *no subquery in subqueries*(v) *is more general than* $(q(\bar{t}), \delta)$ **then**
        delete from $subqueries(v)$ and $unprocessed\_subqueries_2(v)$ all subqueries less general than $(q(\bar{t}), \delta)$;
        add $(q(\bar{t}), \delta)$ to both $subqueries(v)$ and $unprocessed\_subqueries_2(v)$
else if u *is input_p* **then**
  $\Gamma := \emptyset$;
  foreach $\bar{t} \in D$ **do**
    if $p(\bar{t})$ *and atom*(v) *are unifiable by an mgu* γ **then**
      `add-subquery3`$(p(\bar{t})\gamma, \gamma_{|post\_vars(v)}, \Gamma, succ(v))$
  `transfer3`$(\Gamma, v, succ(v))$

Procedure transfer3(D, u, v) *(continued)* 2/2

else if u *is ans_p* **then** $\mathit{unprocessed\_tuples}(v) := \mathit{unprocessed\_tuples}(v) \cup D$;
else if v *is input_p or ans_p* **then**
    foreach $\bar{t} \in D$ **do**
        let $\bar{t}'$ be a fresh variant of $\bar{t}$;
        if $\bar{t}'$ *is not an instance of any tuple from* $\mathit{tuples}(v)$ **then**
            foreach $\bar{t}'' \in \mathit{tuples}(v)$ **do**
                if $\bar{t}''$ *is an instance of* $\bar{t}'$ **then**
                    delete $\bar{t}''$ from $\mathit{tuples}(v)$;
                    foreach $(v, w) \in E$ **do** delete $\bar{t}''$ from $\mathit{unprocessed}(v, w)$;
            if v *is input_p* **then**
                add $\bar{t}'$ to $\mathit{tuples}(v)$;
                foreach $(v, w) \in E$ **do** add $\bar{t}'$ to $\mathit{unprocessed}(v, w)$;
            else
                add $\bar{t}$ to $\mathit{tuples}(v)$;
                foreach $(v, w) \in E$ **do** add $\bar{t}$ to $\mathit{unprocessed}(v, w)$;
else if v *is* $\mathit{filter}_{i,j}$ *and* $\mathit{kind}(v) = \mathit{extensional}$ *and* $T(v) = \mathit{false}$ **then**
    let $p = \mathit{pred}(v)$ and set $\Gamma := \emptyset$;
    foreach $(q(\bar{t}), \delta) \in D$ **do**
        if `term-depth`$(\mathit{atom}(v)\delta) \leq l$ **then**
            foreach $\bar{t}' \in I(p)$ **do**
                if $\mathit{atom}(v)\delta$ *is unifiable with a fresh variant of* $p(\bar{t}')$ *by an mgu* γ **then**
                    `add-subquery3`$(q(\bar{t})\gamma, (\delta\gamma)_{|\mathit{post\_vars}(v)}, \Gamma, \mathit{succ}(v))$
    `transfer3`$(\Gamma, v, \mathit{succ}(v))$
else if v *is* $\mathit{filter}_{i,j}$ *and (*$\mathit{kind}(v) = \mathit{extensional}$ *and* $T(v) = \mathit{true}$ *or* $\mathit{kind}(v) = \mathit{intensional}$*)* **then**
    foreach $(q(\bar{t}), \delta) \in D$ **do**
        if `term-depth`$(\mathit{atom}(v)\delta) \leq l$ **then**
            if *no subquery in* $\mathit{subqueries}(v)$ *is more general than* $(q(\bar{t}), \delta)$ **then**
                delete from $\mathit{subqueries}(v)$ and $\mathit{unprocessed\_subqueries}(v)$ all subqueries less general than $(q(\bar{t}), \delta)$;
                add $(q(\bar{t}), \delta)$ to both $\mathit{subqueries}(v)$ and $\mathit{unprocessed\_subqueries}(v)$;
                if $\mathit{kind}(v) = \mathit{intensional}$ **then**
                    delete from $\mathit{unprocessed\_subqueries}_2(v)$ all subqueries less general than $(q(\bar{t}), \delta)$;
                    add $(q(\bar{t}), \delta)$ to $\mathit{unprocessed\_subqueries}_2(v)$
else `// v is of the form` $\mathit{post\_filter}_i$
    $\Gamma := \{\bar{t} \mid (q(\bar{t}), \varepsilon) \in D\}$;
    `transfer3`$(\Gamma, v, \mathit{ans\_q})$

Appendix E

Functions and Procedures Used for Algorithm 4

Algorithm 4 (on page 63) repeatedly selects an active edge and fires the operation for the edge and uses the function `active-edge4`(u, v) (on page 119), which returns *true* if the data accumulated in u can be processed to produce some data to transfer through the edge (u, v). If `active-edge4`(u, v) is *true* then the procedure `fire4`(u, v) (on page 120) processes the data accumulated in u that has not been processed before and transfers appropriate data through the edge (u, v). The procedure `fire4` uses the procedures `add-tuple` and `add-subquery` (on page 27) and `transfer4`(D, u, v) (on page 121), which specifies the effects of transferring data D through the edge (u, v) of a QSQN-STR.

Function active-edge4(u, v)

Global data: a QSQN-STR $N = (V, E, T, C)$.

Input: an edge $(u, v) \in E$.

Output: *true* if there is data to transfer through the edge (u, v), and $false$ otherwise.

if u *is* $\mathit{pre\_filter}_i$ *or* $\mathit{post\_filter}_i$ **then return** $false$;
else if u *is* $\mathit{input\_p}$ *or* $\mathit{ans\_p}$ **then return** $\mathit{unprocessed}(u, v) \neq \emptyset$;
else if u *is* $\mathit{filter}_{i,j}$ *and* $\mathit{kind}(u) = \mathit{extensional}$ **then**
    return $T(u) = \mathit{true} \wedge \mathit{unprocessed\_subqueries}(u) \neq \emptyset$
else `//` u `is of the form` $\mathit{filter}_{i,j}$ `and` $\mathit{kind}(u) = \mathit{intensional}$
    let $p = \mathit{pred}(u)$;
    if $v = \mathit{input\_p}$ **then return** $\mathit{unprocessed\_subqueries}_2(u) \neq \emptyset$;
    else if $\mathit{neg}(u) = \mathit{true}$ **then return** $\mathit{unprocessed\_subqueries}(u) \neq \emptyset$;
    else return $\mathit{unprocessed\_subqueries}(u) \neq \emptyset \vee \mathit{unprocessed\_tuples}(u) \neq \emptyset$;

Procedure fire4(u, v)

Global data: a stratified logic program P, an extensional instance I, a QSQN-STR $N = (V, E, T, C)$ of P, and a term-depth bound l.

Input: an edge $(u, v) \in E$ such that `active-edge`(u, v) holds.

if u *is* $input\_p$ *or* ans_p **then**
    `transfer4`$(unprocessed(u, v), u, v)$;
    $unprocessed(u, v) := \emptyset$
else if u *is* $filter_{i,j}$ *and* $kind(u) = extensional$ *and* $T(u) = true$ **then**
    let $p = pred(u)$ and set $\Gamma := \emptyset$;
    foreach $(\bar{t}, \delta) \in unprocessed\_subqueries(u)$ **do**
        if $neg(u) = false$ **then**
            foreach $\bar{t}' \in I(p)$ **do**
                if $atom(u)\delta$ *is unifiable with a fresh variant of* $p(\bar{t}')$ *by an mgu* γ **then**
                    `add-subquery`$(\bar{t}\gamma, (\delta\gamma)_{|post\_vars(u)}, \Gamma, v)$
        else
            if $atom(u)\delta \notin \{p(\bar{t}') \mid \bar{t}' \in I(p)\}$ **then**
                `add-subquery`$(\bar{t}, \delta_{|post\_vars(u)}, \Gamma, v)$
    $unprocessed\_subqueries(u) := \emptyset$;
    `transfer4`(Γ, u, v)
else if u *is* $filter_{i,j}$ *and* $kind(u) = intensional$ **then**
    let $p = pred(u)$ and set $\Gamma := \emptyset$;
    if $v = input\_p$ **then**
        foreach $(\bar{t}, \delta) \in unprocessed\_subqueries_2(u)$ **do** let $p(\bar{t}') = atom(u)\delta$, `add-tuple`$(\bar{t}', \Gamma)$;
        $unprocessed\_subqueries_2(u) := \emptyset$;
    else
        foreach $(\bar{t}, \delta) \in unprocessed\_subqueries(u)$ **do**
            if $neg(u) = false$ **then**
                foreach $\bar{t}' \in tuples(ans\_p)$ **do**
                    if $atom(u)\delta$ *is unifiable with a fresh variant of* $p(\bar{t}')$ *by an mgu* γ **then**
                        `add-subquery`$(\bar{t}\gamma, (\delta\gamma)_{|post\_vars(u)}, \Gamma, v)$
            else
                if $atom(u)\delta \notin \{p(\bar{t}') \mid \bar{t}' \in tuples(ans\_p)\}$ **then**
                    `add-subquery`$(\bar{t}, \delta_{|post\_vars(u)}, \Gamma, v)$
        $unprocessed\_subqueries(u) := \emptyset$;
        if $neg(u) = false$ **then**
            foreach $\bar{t} \in unprocessed\_tuples(u)$ **do**
                foreach $(\bar{t}', \delta) \in subqueries(u)$ **do**
                    if $atom(u)\delta$ *is unifiable with a fresh variant of* $p(\bar{t})$ *by an mgu* γ **then**
                        `add-subquery`$(\bar{t}'\gamma, (\delta\gamma)_{|post\_vars(u)}, \Gamma, v)$
            $unprocessed\_tuples(u) := \emptyset$
    `transfer4`(Γ, u, v)

Procedure transfer4(D, u, v)

Global data: a stratified logic program, an extensional instance I, a QSQN-STR $N = (V, E, T, C)$ of P, and a term-depth bound l.
Input: data D to transfer through the edge $(u, v) \in E$.

```
if D = ∅ then return;
if u is input_p then
    Γ := ∅;
    foreach t̄ ∈ D do
        if p(t̄) and atom(v) are unifiable by an mgu γ then
            add-subquery(t̄γ, γ|post_vars(v), Γ, succ(v))
    transfer4(Γ, v, succ(v))
else if u is ans_p then unprocessed_tuples(v) := unprocessed_tuples(v) ∪ D;
else if v is input_p or ans_p then
    foreach t̄ ∈ D do
        let t̄' be a fresh variant of t̄;
        if t̄' is not an instance of any tuple from tuples(v) then
            foreach t̄'' ∈ tuples(v) do
                if t̄'' is an instance of t̄' then
                    delete t̄'' from tuples(v);
                    foreach (v, w) ∈ E do delete t̄'' from unprocessed(v, w);
            if v is input_p then
                add t̄' to tuples(v);
                foreach (v, w) ∈ E do add t̄' to unprocessed(v, w);
            else
                add t̄ to tuples(v);
                foreach (v, w) ∈ E do add t̄ to unprocessed(v, w);
else if v is filter_{i,j} and kind(v) = extensional and T(v) = false then
    let p = pred(v) and set Γ := ∅;
    foreach (t̄, δ) ∈ D do
        if term-depth(atom(v)δ) ≤ l then
            if neg(v) = false then
                foreach t̄' ∈ I(p) do
                    if atom(v)δ is unifiable with a fresh variant of p(t̄') by an mgu γ then
                        add-subquery(t̄γ, (δγ)|post_vars(v), Γ, succ(v))
            else
                if atom(v)δ ∉ {p(t̄') | t̄' ∈ I(p)} then
                    add-subquery(t̄, δ|post_vars(v), Γ, succ(v))
    transfer4(Γ, v, succ(v))
else if v is filter_{i,j} and (kind(v) = extensional and T(v) = true or kind(v) = intensional) then
    foreach (t̄, δ) ∈ D do
        if term-depth(atom(v)δ) ≤ l then
            if no subquery in subqueries(v) is more general than (t̄, δ) then
                delete from subqueries(v) all subqueries less general than (t̄, δ);
                delete from unprocessed_subqueries(v) all subqueries less general than (t̄, δ);
                add (t̄, δ) to both subqueries(v) and unprocessed_subqueries(v);
                if kind(v) = intensional then
                    delete from unprocessed_subqueries_2(v) all subqueries less general than (t̄, δ);
                    add (t̄, δ) to unprocessed_subqueries_2(v)
else // v is of the form post_filter_i
    Γ := {t̄ | (t̄, ε) ∈ D};
    transfer4(Γ, v, succ(v))
```

Bibliography

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison Wesley, 1995.

[2] K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. *Found. of Deductive Databases and Logic Programming*, pages 89–148, 1988.

[3] K.R. Apt. *From Logic Programming to Prolog.* Prentice-Hall, 1997.

[4] K.R. Apt and R. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19:9–71, 1994.

[5] J. Bailey, F. Bry, T. Furche, and S. Schaffert. Semantic Web query languages. In *Encyclopedia of Database Systems*, pages 2583–2586. Springer, 2009.

[6] I. Balbin, G. S. Port, K. Ramamohanarao, and K. Meenakshi. Efficient bottom-up computation of queries on stratified databases. *J. Log. Program.*, 11(3-4):295–344, 1991.

[7] F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of PODS'1986*, pages 1–15. ACM, 1986.

[8] C. Beeri and R. Ramakrishnan. On the power of magic. *J. Log. Program.*, 10:255–299, 1991.

[9] F. Bry. Query evaluation in deductive databases: Bottom-up and top-down reconciled. *Data Knowl. Eng.*, 5:289–312, 1990.

[10] F. Bry, T. Furche, C. Ley, B. Marnette, B. Linse, and S. Schaffert. Datalog relaunched: Simulation unification and value invention. In *Proceedings of Datalog'2010*, volume 6702 of *LNCS*, pages 321–350. Springer, 2010.

[11] S.T. Cao. On the efficiency of query-subquery nets: An experimental point of view. In *Proceedings of SoICT'2013*, pages 148–157. ACM, 2013.

[12] S.T. Cao. A revised version of the proofs of soundness, completeness and data complexity for query-subquery nets. Available at http://mimuw.edu.pl/~sonct/QSQN-proofs.pdf, 2015.

[13] S.T. Cao. An implementation in Java of the evaluation methods QSQN, QSQN-TRE, QSQN-rTRE, QSQN-STR, QSQR and Magic-Sets. Available at http://mimuw.edu.pl/~sonct/EvaluationMethods.zip, 2015.

[14] S.T. Cao. On the efficiency of query-subquery nets with right/tail-recursion elimination in evaluating queries to Horn knowledge bases. In *Proceedings of ICCSAMA'2015*, volume 358 of *Advances in Intelligent Systems and Computing*, pages 243–254. Springer, 2015.

[15] S.T. Cao. Query-subquery nets with stratified negation. In *Proceedings of ICCSAMA'2015*, volume 358 of *Advances in Intelligent Systems and Computing*, pages 355–366. Springer, 2015.

[16] S.T. Cao and L.A. Nguyen. An improved depth-first control strategy for query-subquery nets in evaluating queries to Horn knowledge bases. In *Proceedings of ICCSAMA'2014*, volume 282 of *Advances in Intelligent Systems and Computing*, pages 281–295. Springer, 2014.

[17] S.T. Cao and L.A. Nguyen. An empirical approach to query-subquery nets with tail-recursion elimination. In *New Trends in Database and Information Systems II, selected papers of ADBIS'2014*, volume 312 of *Advances in Intelligent Systems and Computing*, pages 109–120. Springer, 2015.

[18] S.T. Cao, L.A. Nguyen, and A. Szalas. On the Web ontology rule language OWL 2 RL. In P. Jedrzejowicz, N.T. Nguyen, and K. Hoang, editors, *Proceedings of ICCCI'2011*, volume 6922 of *LNCS*, pages 254–264. Springer, 2011.

[19] S.T. Cao, L.A. Nguyen, and A. Szalas. WORL: A Web ontology rule language. In *Proceedings of KSE'2011*, pages 32–39. IEEE, 2011.

[20] S.T. Cao, L.A. Nguyen, and A. Szalas. The Web ontology rule language OWL 2 RL+ and its extensions. *T. Computational Collective Intelligence*, 13:152–175, 2014.

[21] S.T. Cao, L.A. Nguyen, and A. Szalas. WORL: a nonmonotonic rule language for the Semantic Web. *Vietnam J. Computer Science*, 1(1):57–69, 2014.

[22] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *Transactions on Knowledge and Data Engineering (IEEE)*, 1(1):146–166, 1989.

[23] W. Chen, T. Swift, and D.S. Warren. Efficient top-down computation of queries under the well-founded semantics. *J. Log. Program.*, 24(3):161–199, 1995.

[24] K.L. Clark. Predicate logic as a computational formalism. Research Report DOC 79/59, Department of Computing, Imperial College, 1979.

[25] V.S. Costa and D. Vaz. BigYAP: Exo-compilation meets UDI. *Theory and Practice of Logic Programming*, 13:799–813, 2013.

[26] W. Drabent and J. Maluszynski. Well-founded semantics for hybrid rules. In *Proceedings of RR'2007*, volume 4524 of *LNCS*, pages 1–15. Springer, 2007.

[27] T. Eiter, G. Ianni, T. Lukasiewicz, and R. Schindlauer. Well-founded semantics for description logic programs in the Semantic Web. *ACM Trans. Comput. Log.*, 12(2):11, 2011.

[28] J. Freire, T. Swift, and D.S. Warren. Taking I/O seriously: Resolution reconsidered for disk. In L. Naish, editor, *Proc. of ICLP'1997*, pages 198–212. MIT Press, 1997.

[29] A.V. Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):619–649, 1991.

[30] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of Logic Programming Symposium*, pages 1070–1080. MIT Press, 1988.

[31] J. Grant and J. Minker. Deductive database theories. *Knowledge Eng. Review*, 4(3):267–304, 1989.

[32] T.J. Green, S.S. Huang, B.T. Loo, and W. Zhou. Datalog and recursive query processing. *Foundations and Trends in Databases*, 5(2):105–195, 2013.

[33] S.S. Huang, T.J. Green, and B.T. Loo. Datalog and emerging applications: an interactive tutorial. In *Proceedings of SIGMOD'2011*, pages 1213–1216. ACM, 2011.

[34] D.B. Kemp, D. Srivastava, and P.J. Stuckey. Bottom-up evaluation and query optimization of well-founded models. *Theor. Comput. Sci.*, 146(1&2):145–184, 1995.

[35] J.M. Kerisit and J.M. Pugin. Efficient query answering on stratified databases. In *Proceedings of FGCS'88*, pages 719–726, 1988.

[36] M. Knorr, J.J. Alferes, and P. Hitzler. A coherent well-founded model for hybrid MKNF knowledge bases. In *Proceedings of ECAI'2008*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 99–103. IOS Press, 2008.

[37] J.W. Lloyd. *Foundations of Logic Programming, 2nd Edition.* Springer, 1987.

[38] E. Madalińska-Bugaj and L.A. Nguyen. Generalizing the QSQR evaluation method for Horn knowledge bases. In N.T. Nguyen and R. Katarzyniak, editors, *New Challenges in Applied Intelligence Technologies*, volume 134 of *Studies in Computational Intelligence*, pages 145–154. Springer, 2008.

[39] E. Madalińska-Bugaj and L.A. Nguyen. A generalized QSQR evaluation method for Horn knowledge bases. *ACM Trans. on Computational Logic*, 13(4):32, 2012.

[40] B. Motik and R. Rosati. Reconciling description logics and rules. *J. ACM*, 57(5), 2010.

[41] S.A. Naqvi. A logic for negation in database system. In *Workshop on Foundations of Deductive Databases and Logic Programming*, pages 378–387, 1986.

[42] J.F. Naughton, R. Ramakrishnan, Y. Sagiv, and J.D. Ullman. Argument reduction by factoring. *Theor. Comput. Sci.*, 146(1&2):269–310, 1995.

[43] W. Nejdl. Recursive strategies for answering recursive queries - the RQA/FQI strategy. In P.M. Stocker, W. Kent, and P. Hammersley, editors, *Proceedings of VLDB'87*, pages 43–50, 1987.

[44] L.A. Nguyen. An implementation in Prolog of the generalized QSQR evaluation method for Horn knowledge bases. Available at http://www.mimuw.edu.pl/~nguyen/GQSQR-PL.zip, 2011.

[45] L.A. Nguyen and S.T. Cao. A preliminary version of the paper "Query-Subquery Nets". Available at http://arxiv.org/abs/1201.2564, 2012.

[46] L.A. Nguyen and S.T. Cao. Query-subquery nets. In *Proceedings of ICCCI'2012*, LNCS, vol. 7635, pages 239–248. Springer-Verlag, 2012.

[47] U. Nilsson and J. Maluszynski. *Logic, Programming and Prolog.* John Wiley & Sons, Inc., 2nd edition, 1995.

[48] R. Ramakrishnan, C. Beeri, and R. Krishnamurthy. Optimizing existential Datalog queries. In *Proceedings of PODS'1988*, pages 89–102. ACM, 1988.

[49] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Efficient bottom-up evaluation of logic programs. In J. Vandewalle, editor, The State of the Art in Computer Systems and Software Engineering. Kluwer Academic Publishers, 1992.

[50] R. Ramakrishnan and J.D. Ullman. A survey of deductive database systems. *J. Log. Program.*, 23(2):125–149, 1995.

[51] K. Ramamohanarao and J. Harland. An introduction to deductive database languages and systems. *The VLDB Journal*, 3(2):107–122, 1994.

[52] R. Rosati. DL+log: Tight integration of description logics and disjunctive Datalog. In *Proceedings of KR'2006*, pages 68–78. AAAI Press, 2006.

[53] K.A. Ross. Tail recursion elimination in deductive databases. *ACM Trans. Database Syst.*, 21(2):208–237, 1996.

[54] E. Ruckhaus, E. Ruiz, and M.E. Vidal. Query evaluation and optimization in the Semantic Web. *Theory Pract. Log. Program.*, 8(3):393–409, 2008.

[55] D. Saccà and C. Zaniolo. The generalized counting method for recursive logic queries. *Theor. Comput. Sci.*, 62(1-2):187–220, 1988.

[56] F. Sáenz-Pérez at. al. Datalog educational system: A deductive database system. Available at http://des.sourceforge.net, 2014.

[57] K.F. Sagonas and T. Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM Trans. Program. Lang. Syst.*, 20(3):586–634, 1998.

[58] K.F. Sagonas, T. Swift, and D.S. Warren. XSB as an efficient deductive database engine. In R.T. Snodgrass and M. Winslett, editors, *Proceedings of the 1994 ACM SIGMOD Conference on Management of Data*, pages 442–453. ACM Press, 1994.

[59] H. Seki. On the power of Alexander templates. In *Proceedings of PODS'89*, pages 150–159. ACM, 1989.

[60] Y.D. Shen, L.Y. Yuan, J.H. You, and N.F. Zhou. Linear tabulated resolution based on Prolog control strategy. *TPLP*, 1(1):71–103, 2001.

[61] D. Srivastava, S. Sudarshan, R. Ramakrishnan, and J.F. Naughton. Space optimization in deductive databases. *ACM Trans. Database Syst.*, 20(4):472–516, 1995.

[62] S. Staab. Completeness of the SLD-resolution. Slides of a course on advanced data modeling. Available at http://www.uni-koblenz.de/FB4/Institutes/IFI/AGStaab/Teaching/SS08/adm08/DB2-SS08-Slides9.ppt, 2008.

[63] R.F. Stärk. A direct proof for the completeness of SLD-resolution. In E. Börger, H.K. Büning, and M.M. Richter, editors, *Proceedings of CSL'89*, volume 440 of *LNCS*, pages 382–383. Springer, 1990.

[64] P.J. Stuckey and S. Sudarshan. Well-founded ordered search: Goal-directed bottom-up evaluation of well-founded models. *J. Log. Program.*, 32(3):171–205, 1997.

[65] S. Sudarshan, D. Srivastava, R. Ramakrishnan, and J.F. Naughton. Space optimization in the bottom-up evaluation of logic programs. In *Proceedings of SIGMOD'1991*, pages 68–77. ACM Press, 1991.

[66] I. Tachmazidis, G. Antoniou, and W. Faber. Efficient computation of the well-founded semantics over big data. *Theory and Practice of Logic Programming*, 14:445–459, 2014.

[67] H. Tamaki and T. Sato. OLD resolution with tabulation. In E.Y. Shapiro, editor, *Proceedings of ICLP'1986, LNCS 225*, pages 84–98. Springer, 1986.

[68] Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems, Vol. I.* Computer Science Press, Inc., New York, NY, USA, 1988.

[69] L. Vieille. Recursive axioms in deductive databases: The query/subquery approach. In *Proceedings of Expert Database Systems*, pages 179–193, 1986.

[70] L. Vieille. A database-complete proof procedure based on SLD-resolution. In *Proceedings of ICLP*, pages 74–103, 1987.

[71] L. Vieille. Recursive query processing: The power of logic. *Theor. Comput. Sci.*, 69(1):1–53, 1989.

[72] N.F. Zhou and T. Sato. Efficient fixpoint computation in linear tabling. In *Proceedings of PPDP'2003*, pages 275–283. ACM, 2003.

List of Figures

List of Tables

Index