Evaluating DATALOG Programs over Infinite and Founded Databases

EVELINA ZARIVACH

Evaluating DATALOG Programs over Infinite and Founded Databases

Research Thesis

Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science in Computer Science

EVELINA ZARIVACH

Submitted to the Senate of the Technion – Israel Institute of Technology
Tevet 5768 HAIFA December 2007

The Research Thesis Was Done Under the Supervision of Dr. Yossi Gil in the Faculty of Computer Science, Technion

THE GENEROUS FINANCIAL HELP OF THE TECHNION IS GRATEFULLY ACKNOWLEDGED

Acknowledgments

I would like to express my appreciation to people without whom this work would not be completed.
I thank my advisor, Assoc. Prof. Yossi Gil, for his enormous help and endless patience. He taught me that perfection has no limits.
I thank Dr. Sara Cohen for her valuable assistance and guidance during my thesis.
I also want to thank Itay Maman for his involvement, his help and encouragement.
Fruitful discussions with Elena Tulchinsky are gratefully acknowledged.
Thanks to my parents and my beloved husband Igor for their support and belief in me.
And last, but not least, I thank my precious daughter Emily, who was born two days after my thesis examination, for waiting for the right moment to show up.

Contents

List of Figures

List of Algorithms

List of Definitions

Abstract

Traditionally, infinite databases were studied as a data model for queries that may contain function symbols (since functions may be expressed as infinite relations). Recently, the interest in infinite databases has been sparked by additional scenarios, e.g., as a formal model of a database of an open-world software or of other relations that may be spread across the Web. In the course of implementing a database system for querying Java software, we found that the universe of Java code can be effectively modeled as an infinite database. This modeling makes it possible to distinguish between queries which are "open-ended," that is, whose result may grow as software components are added into the system, and queries which are "closed," in that their result does not change as the software base grows. Further, closed queries can be implemented much more efficiently than open queries.

This work revisits the weak safety and termination problems for recursive DATALOG programs evaluated over infinite databases. In particular, an algorithm is presented that computes all finiteness constraints for the IDB predicates of a program, given a set of finiteness constraints over the EDB predicates. In addition to being of interest in itself, this algorithm also presents an alternative method to check for weak safety and as a skeleton for query evaluation. A sufficient condition for program termination is also presented, provided that the program and database satisfy certain natural constraints. These constraints are often satisfied in the context of software analysis problems. For programs that satisfy these constraints, we also provide an algorithm to generate an efficient evaluation scheme of closed queries, which is a generalization of Vieille's famous QSQR algorithm for top-down evaluation of Datalog programs. A by-product of this work is a rather terse and elegant representation of QSQR.

List of Symbols

Symbol	Denotes
$\mathcal{A}(r)$	set of adapted rules corresponding to a rule r
a	atom
$\mathcal{C}$	set of finiteness constraints
$\mathcal{C}_\Pi$	set of constraints on IDB predicates implied by Π
$\mathfrak{C}(\mathcal{F})$	set of all constraints that $\mathcal{F}$ satisfy
$\mathcal{D}$	database
$\mathbb{D}$	domain
$\mathcal{F}$	set of facts for EDB and possible IDB predicates
p	predicate
p_i	the i^{th} position of predicate p
$p^{\mathbf{x},\mathbf{y}}$	auxiliary predicate of p, $\mathbf{x}$, $\mathbf{y} \subseteq \mathbf{pos}(p)$
q	goal predicate
r	rule
$r^{\mathbf{x},\mathbf{y}}$	adapted rule of r which defines $p^{\mathbf{x},\mathbf{y}}$
$\vec{\boldsymbol{r}}$	sequence of rules
$\mathbf{X}^+_{\mathcal{C},r}$	closure of a set $\mathbf{X} \subseteq \mathbf{terms}(r)$ with respect to $\mathcal{C}$ and rule r
$\ldots$, **X**, **Y**, **Z**	sets of variables (upper-case boldface letters)
$\ldots$, x, y, z	variables (lower-case letters from the end of the alphabet)
$\ldots$, **x**, **y**, **z**	sets of positions (lower-case boldface letters)
γ	expansion rule
Π	DATALOG program
$\widetilde{\Pi}$	adapted program of Π

Symbol	Denotes
$\Pi^i(\mathcal{F})$	the result of applying all rule sequences of Π on $\mathcal{F}$ where each sequence is no longer than i
$\Pi^i_p(\mathcal{F})$	restriction of a set of facts to facts on predicate p
σ	finiteness constraint
τ	rule
$\mathrm{ar}(p)$	arity of predicate p
body(r)	body of rule r
cls$(r, \mathbf{x}, \mathcal{C})$	closure sequence of r with respect to **x** and $\mathcal{C}$
consts(a)	set of all constants appearing in a
consts(Π)	set of constants appearing in any rule of Π
$\mathbf{edb}(\Pi)$	EDB predicates of Π
$\mathrm{head}(r)$	head atom of rule r
$\mathbf{idb}(\Pi)$	IDB predicates of Π
$\mathsf{min}(\mathcal{C})$	minimized version of $\mathcal{C}$
pos(p)	set of positions of p, i.e., $\{p_1, \ldots, p_{\mathrm{ar}(p)}\}$
$\mathrm{pred}(a)$	predicate of an atom a
$\mathrm{pred}(r)$	predicate of the head of rule r, i.e., $\mathrm{pred}(r) = \mathrm{pred}(\mathrm{head}(r))$
terms(a)	set of all terms appearing in a
vars(a)	set of all variables appearing in a

Chapter 1

Introduction

Usually, a database contains relations of finite size. However, there are natural settings which can be better modeled by infinite databases, over which a set of finiteness constraints is defined. In this research we are interested in the safety problem, which is one of most fundamental issues related to DATALOG [4] programs over infinite relations. We say that a DATALOG program is *safe*, if it yields a finite result over all databases which satisfy some given finiteness constraints. Instead of studying the safety problem directly, we consider the *weak safety* and *termination* properties. Intuitively, a program (i) is weakly safe if it yields a finite answer for all finite applications of its rules and, (ii) terminates if every sequence of rule applications eventually ceases to yield new results.

To motivate our study of infinite databases, we describe below three different scenarios which involve (some degree of) inherent infiniteness of the database.

1.1 Function Symbols

Classically, infinite databases were first introduced as an abstraction that allow programs with function symbols to be modeled as function-free programs over infinite relations. As an intuitive example, consider the following DATALOG program which contains two function symbols:

$$\mathsf{q}(\mathtt{x}+1) \leftarrow \mathsf{p}(\mathtt{x}).$$
$$\mathsf{q}(\mathtt{x}) \leftarrow \mathsf{q}(\sqrt{\mathtt{x}}), \mathsf{p}(\mathtt{x}).$$

As [21] showed, this program can be abstractly modeled as follows, where succ and sqrt are infinite relations.

$$\mathsf{q}(\mathtt{y}) \leftarrow \mathsf{p}(\mathtt{x}), \mathsf{succ}(\mathtt{x}, \mathtt{y}).$$
$$\mathsf{q}(\mathtt{x}) \leftarrow \mathsf{q}(\mathtt{y}), \mathsf{p}(\mathtt{x}), \mathsf{sqrt}(\mathtt{x}, \mathtt{y}).$$

Finiteness constraints were introduced in [21] to model known characteristics of the function symbols, such that for each x, there are finitely many y such that $\mathsf{sqrt}(\mathtt{x}, \mathtt{y})$ holds.

1.2 Open-World Software

Recently, the interest in infinite databases has been sparked by additional scenarios, e.g., as a formal model of a database of an open-world software or of other relations that may be spread across the Web. Open-world software is infinite in the sense that it is constantly growing, and thus, cannot be completely explored at any moment in time. For example, given a class C, there may be an unbounded number of program classes that *inherit* from C, that *call a method* from C, or that *have as a data member* an instance of C. Thus, querying in the open-world software scenario is naturally modeled as querying over infinite relations. This specific domain also gives rise to finiteness constraints, e.g., a class may inherit only from a finite number of classes.

Our study of infinite databases was motivated by JTL [5], a new DATALOG based system for making queries over software, which uses infinite relations as its data model. As an example of the usefulness of this paradigm for querying JAVA software, consider the following JTL query, which finds (i) all public interfaces or (ii) all public interfaces or classes that extend an abstract class or interface.

```
public [extends T, T abstract | interface];
```

The DATALOG program equivalent to the above is as follows.

$$\mathsf{q}(\mathtt{x}) \leftarrow \mathsf{public}(\mathtt{x}), \mathsf{p}(\mathtt{x}).$$
$$\mathsf{p}(\mathtt{x}) \leftarrow \mathsf{interface}(\mathtt{x}).$$
$$\mathsf{p}(\mathtt{x}) \leftarrow \mathsf{extends}(\mathtt{x}, \mathtt{y}), \mathsf{abstract}(\mathtt{y}).$$

Note that public, interface, abstract and extends are EDB predicates. Although this program is nonrecursive, it is also possible to express recursive programs in JTL.

Even in an open-environment, finiteness constraints have a natural manifestation. For example, the transitive closure of a "uses" relationship between programs is assumed to be bounded. In other words, the programming model is such that, it is unknown which classes may be using a given class, and in general, the number of these classes is unbounded. However, the list of classes that the given class uses, directly or indirectly, is bounded, and must be available to the compiler at compile time. (This assumption is critical, as it allows a program to be compiled and executed by dynamically loading required components.)

The software-engineering research community has regained interest in applying the logic paradigm for implementing algorithms for processing software, and more generally, frameworks for developing such algorithms. Prime examples of this interest include the CodeQuest system for formulating queries on code [10], ALPHA [19], and JQuery [11]. Such systems are not limited to simple queries, but also e.g., advance frameworks for dataflow algorithms [18] for developing such algorithms, and optimization schemes for algorithms implemented in this framework [28], as well as efficient implementation of specific algorithms [29]. Some less recent work for using the logic paradigm, and in particular DATALOG, for these tasks include [7,22], as well as the *ASTLog* framework [6], the XL C++ browser [12], and many more. Thus, the results in this framework can be applied to additional software engineering systems, beyond JTL.

1.3 Access Constraints

Infinite databases can also be used as an abstraction for computation that is highly inefficient. Consider, for example, a predicate $\mathsf{tree}(\mathtt{x},\mathtt{y})$, which holds pairs of parent-child node ids in a tree structure. It may be the case that given a value for x, it is easy to compute all values for y (since we have forward pointers), yet given a value for y, it is very inefficient to compute x (if we do not store backward pointers). Such constraints have been modeled in the past as *access constraints* (sometimes called *binding patterns*) and the rewriting problem for queries with access constraints has been extensively studied, e.g., [8, 9, 16]. An alternative modeling of such scenarios is to consider tree as an infinite relation, with a finiteness constraint that specifies the manner(s) in which it can be efficiently accessed.

We do not discuss the exact similarities and differences between these two alternative

models. However, it is of interest to note that our results shed some light on problems related to querying with access constraints. For example, our algorithm for implication of finiteness constraints can be adapted to imply access constraints over IDB predicates, when given access constraints over the EDB predicates.

1.4 Related Work

The problem of deciding safety (i.e., finiteness of results) of a DATALOG program has been extensively studied. Safety of recursive DATALOG programs without function symbols, but with negation, is known to be undecidable [20, 25]. Safety is also undecidable for DATALOG programs with function symbols [24]. This latter result motivated [21] to abstractly model DATALOG programs with function symbols as function-free programs over infinite relations. [21] also introduced finiteness constraints to model known characteristics of function symbols.

The safety problem for DATALOG programs over infinite relations, with finiteness constraints, was studied in [23]. In particular, [23] showed that safety can be reduced to a combination of two properties: *weak safety* (i.e., finiteness of results for every finite number of rule applications) and termination. They presented a method to determine weak safety, and showed that termination is undecidable. For monadic programs, [23] proved that safety can be determined in polynomial time.

Several stronger notions than safety have also been studied for programs over infinite relations. Supersafety was considered in [13, 14] and shown to be decidable. Supersafety is a sufficient, but not necessary, condition for safety. Intuitively, supersafety requires finiteness of results in all fixpoint models, whereas safety requires finiteness of results only in the least fixpoint model. A variant characteristic, called strong safety, was studied in [15]. Basically, a program is strongly safe if all intermediate rules (and not only the goal predicate) yield finite results. For a special case, [15] showed how to evaluate all results for a strongly safe program, using a bottom-up computation. One of the requirements in [15] is that each rule can be computed in a left-to-right ordering of its atoms, such that the variables in a specific atom are bounded by those appearing to its left. Our results can also be used to check such properties of rules.

1.5 Contributions

This work presents new results on the safety problem for DATALOG programs over infinite relations. Our contributions can be summarized as follows.

- We present an algorithm (Chapter 4 and Chapter 5) to determine finiteness constraints on IDB predicates defined in a DATALOG program based on the finiteness constraints defined on the EDB predicates. Our algorithm finds all finiteness constraints that must hold on the IDB after any finite number of rule applications. This result is useful in itself since it gives us insight on the characteristics of the IDB predicates, which can be important for developing query computation algorithms, such as the type in [15].

- We present an alternative characterization, based on this algorithm, of DATALOG programs which are *weakly safe* (Chapter 6).

- The termination problem is also considered (Chapter 7). Our EDB predicates can be binary (as opposed to the monadic predicates considered in [23]). We decide termination when the database is *founded* which is natural, in particular, in the software model.

- A characterization of DATALOG programs which can be evaluated even if they require in their evaluation partial exploration of infinite values is presented (Chapter 8). (This is the case if the program needs to check e.g., if a given class has at least one class that inherits from it.) An actual evaluation algorithm, based on the famous Vieille's [26, 27] query-subquery top-down evaluation technique is presented in Chapter 9. (We believe that our presentation of the algorithm is a bit more elegant and easy to understand than the original formulation.)

Chapter 2

Preliminaries

This section briefly reviews the basic syntax and semantics of positive, recursive DATALOG programs, which are evaluated over a possibly infinite database. This review is necessary in order to introduce the notation that we will be using throughout the research.

2.1 Syntax

Relations in DATALOG are represented by *predicates*, and are abstractly denoted with p and q. We use $\mathrm{ar}(p)$ to denote the arity of the predicate p. When discussing concrete examples of predicates, we will use the sanserif font, e.g., members, parent. For a predicate p, we denote by $p_1, p_2, \ldots$ its positions.

Let $\mathbb{V}$ be an enumerable set of *variable* symbols and $\mathbb{D}$ be an enumerable set of *constant* symbols. We shall use lower-case letters from the end of the Latin alphabet, i.e., x, y, z, etc., to denote variables and upper-case bold letters to denote sets of variables **X**, **Y**, **Z**. Constants are quoted, e.g., 'Moses', 'Isaac', etc. *Terms* are either constants or variables and are denoted t, t_1, t_2, etc.

An *atom* a is of the form $p(t_1, \ldots, t_n)$ where p is a predicate symbol of arity n and each t_i is a term. We use $\mathrm{pred}(a)$ to denote the predicate of a and we use $\mathrm{ar}(a)$ as a shorthand notation for $\mathrm{ar}(\mathrm{pred}(a))$. For an atom a, we denote by $t_i(a)$ the term which appears at position i. Terms which are mapped to a constant are said to be *bound*; other terms are *free*. In parent(x, 'Moses'), the first term is free while the second is bound. A *fact* is a ground atom, i.e., an atom in which all arguments are bound. For example, parent('Amram', 'Moses') is a fact. The phrase p-*fact* refers to a fact a such

that $\mathrm{pred}(a) = p$.

Let **vars**(a) (respectively **consts**(a)) denote the set of all variables (respectively constants) appearing in atom a. Let **terms**(a) = **vars**$(a) \cup$ **consts**(a), i.e., **terms**(a) is the set of all the terms appearing in the atom a. An *assignment* is a function $\mu : \mathbb{V} \to \mathbb{D}$. By applying an assignment μ to an atom a, one derives a $\mathrm{pred}(a)$-fact.

A *rule* r has the form $p(t_1, \ldots, t_k) \leftarrow a_1, \ldots, a_n$, where $p(t_1, \ldots, t_k)$ is the *head* of r, and $a_1, \ldots, a_n$ is the *body* of r. We use the overloaded notation $\mathrm{pred}(r)$ to denote the predicate of the head of r. If $p = \mathrm{pred}(r)$, we say that r *defines* p. We overload the notations **vars**(r), **consts**(r) and **terms**(r) to represent all the variables, constants and terms(respectively) appearing in rule r. For $i = 1, 2, \ldots,$ let **terms**$_i(r)$ be the i^{th} element of **terms**(r) in some enumeration of this set. In Figure 2.1, **terms**$(\tau_6) = \{\mathtt{x}, \mathtt{y}, \mathtt{w}, \text{'Bill'}\}$.

A DATALOG *program* Π is a *finite* collection of rules, with a designated predicate, called the *goal*. We use **consts**(Π) to denote the set of constants appearing in any rule of Π, i.e., **consts**$(\Pi) = \bigcup_{r \in \Pi}$ **consts**(r) . We distinguish between two kinds of predicates that appear in a program: (i) extensional database (*EDB*) predicates, denoted $\mathbf{edb}(\Pi)$, which are predicates that do not occur in the head of any of the program's rules, and (ii) intensional database (*IDB*) predicates (all other predicates), denoted $\mathbf{idb}(\Pi)$. By convention, we use q to denote the goal of a program. We always require that $q \in \mathbf{idb}(\Pi)$. We assume that the goal predicate is defined by a single rule.

2.2 Semantics

A *database* $\mathcal{D}$ is a possibly *infinite* set of facts. To be exact, for each EDB predicate p, the database $\mathcal{D}$ may contain infinitely many p-facts; $\mathcal{D}$ usually does not contain any facts for the IDB predicates. The *result* of applying a rule r to a database $\mathcal{D}$ is defined in the standard fashion. Informally, the semantics of a rule is "If the body atoms are true then so is the head atom." To make the semantics precise, we consider a set $\mathcal{F}$ that contains facts for the EDB (and possibly for the IDB) predicates. Such sets are intermediate values during the evaluation of a program on a database. Then, an application of a rule r: $p(t_1, \ldots, t_k) \leftarrow a_1, \ldots, a_n$ to $\mathcal{F}$ produces a set of facts denoted $r(\mathcal{F})$, such that *(1)* every fact in $\mathcal{F}$ is in $r(\mathcal{F})$, and *(2)* if μ is an assignment that satisfies the body of r (i.e., $\mu(a_i) \in \mathcal{F}$, for all i), then also $\mu(p(t_1, \ldots, t_k)) \in r(\mathcal{F})$.

For a sequence $\vec{r}$ of rules, let $\vec{r}(\mathcal{F})$ denote the set of all facts obtained by applying

Figure 2.1 A DATALOG program

τ_1: heir(x, y) $\leftarrow$ child(x, y).

τ_2: heir(x, y) $\leftarrow$ child(x, z), heir(z, y).

τ_3: cousins(x, y, z) $\leftarrow$ child(x′, z), child(x, x′),
child(y′, z), child(y, y′).

τ_4: heirs(x, u, v) $\leftarrow$ heir(x, u), heir(x, v), not_eq(u, v).

τ_5: multi(x, y) $\leftarrow$ heirs(x, u, v), cousins(u, v, y),
dependant(x, y).

τ_6: q(x, y) $\leftarrow$ multi(x, y), heir('Bill', x), child(w, 'Bill').

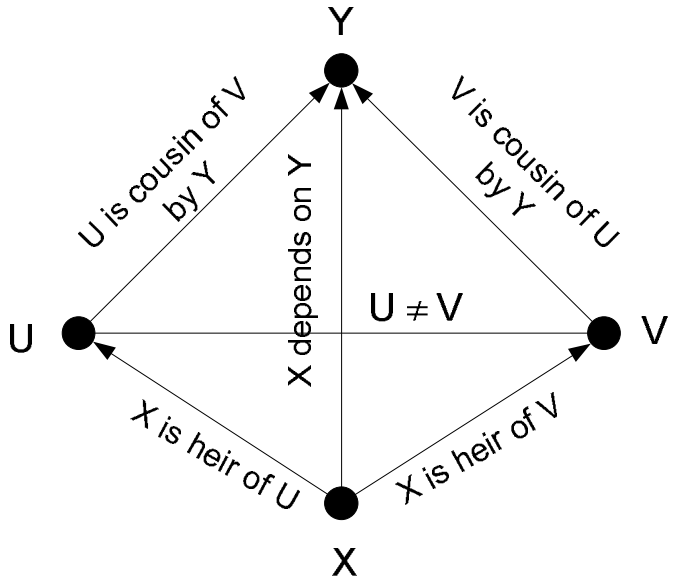
the rules in $\vec{r}$ in sequence to $\mathcal{F}$, i.e., if $\vec{r}$ is empty, then $\vec{r}(\mathcal{F}) = \mathcal{F}$. Otherwise, $\vec{r} = \vec{r}'r$, where $\vec{r}'$ is a sequence and r is a rule, in which case $\vec{r}(\mathcal{F}) = r(\vec{r}'(\mathcal{F}))$. The notation $\Pi^i(\mathcal{F})$ will stand for the union of all $\vec{r}(\mathcal{F})$, where $\vec{r}$ is a sequence of at most i rules selected from Π. Also, let $\Pi^\infty(\mathcal{F}) = \bigcup_{i\geq 0} \Pi^i(\mathcal{F})$.

If p is a predicate, then subscript p will be used to denote the restriction of a set of facts to p-facts only. Thus, $\Pi^i_p(\mathcal{F})$ is the set of p-facts in $\Pi^i(\mathcal{F})$, and $\vec{r}_p(\mathcal{F})$ is defined similarly. The *result* of applying Π to a database $\mathcal{D}$ is $\Pi^\infty_q(\mathcal{D})$ where q is Π's goal. Note that $\Pi^\infty_q(\mathcal{D})$ may be infinite if $\mathcal{D}$ is infinite.

For the purpose of illustration, Figure 2.1 presents a simple DATALOG program, which will be used as the running example of this work. The program is defined over the following EDB predicates: child, dependant and not_eq. In particular, a fact child('c', 'p') states that 'c' is a "child" of 'p' and a fact dependant('a', 'b') represents a fact in which 'a' depends upon 'b'. These predicates can be interpreted over the domain of JAVA classes, with the "uses" semantics. It can also be interpreted over the domain of genealogy of characters in the Bible (or Greek mythology for that matter), with the meaning of "preceding."

The schematic representation of the multi predicate in the program of Figure 2.1 is depicted in Figure 2.2.

Figure 2.2 Representation of multi predicate from Figure 2.1.



2.3 Expansion Rules

We will find it convenient to summarize the application of a rule sequence in a sequence of *expansion* rules, i.e., rules which involve only EDB predicates. We will use γ to denote a single expansion rule and Γ to denote a set of expansion rules. Fac. 2.1 is well known, and follows, e.g., from [17].

Fact 2.1. *For every finite sequence of rules $\vec{r}$ and every predicate p there exists a finite set of expansion rules Γ which uses only the constants occurring in the rules of $\vec{r}$, such that*

$$\vec{r}_p(\mathcal{D}) = \bigcup_{\gamma \in \Gamma} \gamma_p(\mathcal{D})$$

regardless of $\mathcal{D}$.

In our running example, applying τ_1 and then τ_4 is the shortest sequence of rule applications that generates heirs-facts. The expansion rule for this sequence is

$$\begin{aligned} \mathsf{heirs}(\mathtt{x},\mathtt{u},\mathtt{v}) \leftarrow & \mathsf{child}(\mathtt{x},\mathtt{u}), \mathsf{child}(\mathtt{x},\mathtt{v}), \\ & \mathsf{not\_eq}(\mathtt{u},\mathtt{v}). \end{aligned} \tag{2.1}$$

Similarly, one sequence that yields a multi-fact, is by applying first τ_1 and τ_4 (to obtain heirs-facts), then τ_3 (to obtain cousins-facts), and finally τ_5. The corresponding expansion rule is similar to (2.1), but a bit longer

$$\begin{aligned} \mathsf{multi}(\mathtt{x},\mathtt{y}) \leftarrow\ & \mathsf{child}(\mathtt{x},\mathtt{u}), \mathsf{child}(\mathtt{x},\mathtt{v}), \mathsf{not\_eq}(\mathtt{u},\mathtt{v}), \\ & \mathsf{child}(\mathtt{u}',\mathtt{y}), \mathsf{child}(\mathtt{u},\mathtt{u}'), \mathsf{child}(\mathtt{v}',\mathtt{y}), \\ & \mathsf{child}(\mathtt{v},\mathtt{v}'), \mathsf{dependant}(\mathtt{x},\mathtt{y}). \end{aligned} \tag{2.2}$$

Recall that the result of applying a program Π, with goal q, to a database $\mathcal{D}$ is $\Pi_q^\infty(\mathcal{D})$. It follows from Fac. 2.1 that there exists an infinite series of expansion rules $\gamma^1, \gamma^2, \ldots$ defining q such that

$$\Pi_q^\infty(\mathcal{D}) = \bigcup_{i=1}^{\infty} \gamma_q^i(\mathcal{D}). \tag{2.3}$$

Henceforth, we shall tacitly assume that the head atom of any rule r does not contain any variable $\mathtt{v} \in \mathbf{vars}(r)$ more than once and does not contain constants. No generality is lost. Rules can always be brought to this form without changing their semantics by introduction of auxiliary variables and by using the infinite EDB predicate $\mathsf{eq}(\mathtt{x},\mathtt{y})$ which holds whenever $\mathtt{x} = \mathtt{y}$. For example, rule

$$\mathsf{a}(\mathtt{x},\mathtt{x},\mathtt{y},\text{‘Ben’}) \leftarrow \mathsf{a}(\mathtt{x},\text{‘Dan’},\mathtt{z}).$$

will be transformed to

$$\mathsf{a}(\mathtt{x},\mathtt{w},\mathtt{y},\mathtt{u}) \leftarrow \mathsf{a}(\mathtt{x},\text{‘Dan’},\mathtt{z}), \mathsf{eq}(\mathtt{w},\mathtt{x}), \mathsf{eq}(\mathtt{u},\text{‘Ben’}).$$

Chapter 3

The Safety Problem

In this section we present a theory of *finiteness constraints* which is crucial to the analysis of the problem that we research. Informally, the problem is:

> *Given a* DATALOG *program and restrictions over its database, decide whether the result of a program is finite for any infinite database that meets the restrictions.*

One should understand that when there are no constraints on the database, nothing meaningful can be stated about the program's semantics. Even a simple DATALOG program, such as

$$\mathsf{moses\_son}(\mathtt{x}) \leftarrow \mathsf{parent}(\text{`Moses'}, \mathtt{x}).$$

can deduce an infinite number of facts to moses_son predicate when no restrictions are imposed on parent predicate.

Consider, on the other hand, a case in which we restrict the set of parent-facts in the database such that for any 'c' $\in \mathbb{D}$, the set $\{\mathtt{x} \mid \mathsf{parent}(\text{`c'}, \mathtt{x})\}$ is finite. Under this restriction, we can conclude that the above program deduces only finitely many facts to its goal. To express such restrictions, we use *finiteness constraints*, as defined in [21].

Definition 3.1. *Let p be a predicate. Then,* $\mathbf{pos}(p)$ *is the set of symbols* $\{p_1, \ldots, p_{\mathrm{ar}(p)}\}$*, and a* finiteness constraint *(*constraint *for short) of p is an expression of the form* $\mathbf{x} \rightsquigarrow \mathbf{y}$*, where*

$$\mathbf{x}, \mathbf{y} \subseteq \mathbf{pos}(p).$$

A facts' set $\mathcal{F}$ satisfies constraint $\mathbf{x} \rightsquigarrow \mathbf{y}$ if the search for p-facts in $\mathcal{F}$ with some fixed assignment to positions $\mathbf{x}$, yields only a finite variety of combinations of values for positions $\mathbf{y}$. More formally,

Definition 3.2. *Let $\sigma = \mathbf{x} \rightsquigarrow \mathbf{y}$ be a constraint on predicate p and $\mathcal{F}$ be a set of facts. Then, $\mathcal{F} \models \sigma$ (read $\mathcal{F}$* satisfies *σ) if the set*

$$\big\{ b[\mathbf{y}] \;\big|\; b \in \mathcal{F} \text{ and } \operatorname{pred}(b) = p \text{ and } b[\mathbf{x}] = a[\mathbf{x}] \big\}$$

is finite for every p-fact $a \in \mathcal{F}$. If $\mathcal{C}$ is a set of constraints, then $\mathcal{F} \models \mathcal{C}$ if $\mathcal{F} \models \sigma$ for all $\sigma \in \mathcal{C}$.

As an example, consider the ternary predicate intersect, in which a fact intersect('c1', 'c2', 'p') states that 'c1' and 'c2' are two circles intersecting at a point 'p'. Then, (infinite) set of all facts about intersections of distinct circles in the plane satisfies the constraint $\{\mathsf{intersect}_1, \mathsf{intersect}_2\} \rightsquigarrow \{\mathsf{intersect}_3\}$, since there are at most two points in which such circles intersect. This set does not satisfy any other constraints.

Remark 3.3. *Using constraints it is possible to state that the number of p-facts, for some predicate p, must be finite. Formally, this is written as $\emptyset \rightsquigarrow \mathbf{pos}(p)$.*

Remark 3.4. *Note that finiteness constraints are a somewhat weaker version of functional dependencies. Not surprisingly, Armstrong's axioms also characterize finiteness constraints for EDB predicates [21].*

Let $\mathfrak{C}(\mathcal{F})$ denote the set of *all* constraints that set $\mathcal{F}$ satisfies. The following fact is easily shown.

Fact 3.5. *For all finite sequences, $\mathcal{F}_1, \ldots, \mathcal{F}_n$*

$$\mathfrak{C}(\mathcal{F}_1 \cup \cdots \cup \mathcal{F}_n) = \mathfrak{C}(\mathcal{F}_1) \cap \cdots \cap \mathfrak{C}(\mathcal{F}_n) \tag{3.1}$$

(Unfortunately, this fact does not hold for infinite sequences.)

After defining the notion of finiteness constraints, we are ready to formally state the central problem of the research. For the purpose of the following definitions, let Π be a fixed DATALOG program, and let predicate q be its goal. Also let $\mathcal{C}$ be a set of constraints. Then, the main problem of this research is to determine whether the set $\Pi_q^\infty(\mathcal{D})$ is finite whenever $\mathcal{D} \models \mathcal{C}$, i.e., to decide whether a given program is *safe* or not.

Definition 3.6. *Program* Π *is* safe *if* $\Pi_q^\infty(\mathcal{D})$ *is finite whenever* $\mathcal{D} \models \mathcal{C}$.

It has been shown that the safety problem can be reduced [23] to two problems: **(i)** the *weak safety* problem, which is to decide whether any finite sequence of program rule applications yields a finite number of facts to its goal, and **(ii)** the *termination* problem, which is to decide whether there is a finite number of rule applications after which no new facts are added to the program's goal. Formally,

Definition 3.7. *We say that* Π *is* weakly safe *with respect to* $\mathcal{C}$ *if the set* $\Pi_q^n(\mathcal{D})$ *is finite for all* $n \geq 0$ *whenever* $\mathcal{D} \models \mathcal{C}$.

Definition 3.8. *We say that* Π *is* terminating *with respect to* $\mathcal{C}$ *if there exists* $n \geq 0$ *such that* $\Pi_q^\infty(\mathcal{D}) = \Pi_q^n(\mathcal{D})$ *whenever* $\mathcal{D} \models \mathcal{C}$.

Fact 3.9. Π *is safe iff it is both weakly safe and terminating [23].*

It is also known [23] that the weak safety problem is complete for exponential time. However, no algorithm has been shown to, given a program, deduce all constraints for the IDB predicates that follow from the constraints on the EDB predicates, for all finite applications of the program rules. Such an algorithm is the topic of Chapter 4 and of Chapter 5. This algorithm is interesting of itself, since it proves that the finite implication problem for constraints is decidable. It is also useful as an alternative method for determining weak safety (see Chapter 6) and as a skeleton for query evaluation.

Chapter 4

Single Rule Constraints Implication

Let r be a rule, and $\mathcal{C}$ be a set of constraints. This section is concerned with the constraints that can be inferred from $\mathcal{C}$ on the output of a single application of r.

Definition 4.1. *Let σ be a constraint on* $\mathrm{pred}(r)$. *Then, $\mathcal{C} \models_r \sigma$ ($\mathcal{C}$* implies *σ in r) if for every set of facts $\mathcal{F}$, the set $r(\mathcal{F})$ satisfies σ whenever $\mathcal{F} \models \mathcal{C}$.*

Intuitively, $\mathcal{C} \models_r \sigma$ means that if all constraints of $\mathcal{C}$ hold prior to an application of r, then σ holds after a single application of r. Let $\mathcal{C}_r$ denote the set of all constraints implied by $\mathcal{C}$ with respect to rule r, i.e., $\mathcal{C}_r = \{\sigma \,|\, \mathcal{C} \models_r \sigma\}$.

Consider, for example, rule τ_3 in the running example. For constraints set $\mathcal{C} = \{\{\mathsf{child}_1\} \rightsquigarrow \{\mathsf{child}_2\}\}$, we have

$$\mathcal{C}_{\tau_3} = \big\{\, \{\mathsf{cousins}_1\} \rightsquigarrow \{\mathsf{cousins}_3\}\,, \\ \{\mathsf{cousins}_2\} \rightsquigarrow \{\mathsf{cousins}_3\} \,\big\}.$$

Now, a *rule constraint* (or a *causation*) is an expression of the form

$$\mathbf{X} \rightsquigarrow \mathbf{Y}$$

where $\mathbf{X}, \mathbf{Y} \subseteq \mathbf{terms}(r)$. For example, $\{\mathtt{x}\} \rightsquigarrow \{\mathtt{z}\}$ is a causation of the rule τ_3 defined in Figure 2.1.

Inference in the context of rule r, must be done in terms of r's vocabulary, that is the set $\mathbf{terms}(r)$. We introduce mechanisms for vocabulary translation: For a p-atom a, we introduce a function $\mathsf{term2pos}_a(\cdot)$ which given a set of terms of a, returns the equivalent set of p-positions, e.g., for $a = \mathsf{p}(\mathtt{x}, \text{'B'}, \mathtt{y}, \mathtt{x}, \text{'B'}, \mathtt{y})$,

$$\mathsf{term2pos}_a(\{\mathtt{x}, \text{'B'}\}) = \{\mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_4, \mathsf{p}_5\}\,.$$

Function $\mathsf{pos2term}_a(\cdot)$ is simply $\mathsf{term2pos}_a{}^{-1}(\cdot)$. Also, for p-rule r with head atom h, define function $\mathsf{term2pos}_r(\mathbf{X})$, where $\mathbf{X} \subseteq \mathbf{terms}(r)$, as

$$\mathsf{term2pos}_h(\mathbf{X} \cap \mathbf{terms}(h)),$$

that is, convert to p-positions only terms occurring in the head. Function $\mathsf{pos2term}_r(\cdot)$ is simply $\mathsf{pos2term}_h(\cdot)$.

To define the semantics of a causation $\sigma = \mathbf{X} \rightsquigarrow \mathbf{Y}$, we construct a (predicate-) constraint

$$\sigma' = \mathsf{term2pos}_{r'}(\mathbf{X}) \rightsquigarrow \mathsf{term2pos}_{r'}(\mathbf{Y})$$

where rule r' is constructed from r by selecting p', a fresh predicate symbol not occurring in r, and letting r' be the p'-rule identical to r except that all members of $\mathbf{terms}(r)$ occur in its head term, i.e., the bodies of r' and r are the same, and the head of r' is

$$h' = p'(\mathbf{terms}_1(r), \ldots, \mathbf{terms}_k(r)),$$

where $k = |\,\mathbf{terms}(r)|$. We write $\mathcal{C} \models_r \sigma$ (read $\mathcal{C}$ *implies* σ in rule r), or simply $\mathcal{C} \models \sigma$ (read $\mathcal{C}$ *implies* σ) when the rule is clear from context, iff $\mathcal{C} \models_{r'} \sigma'$.

To determine logical implications of causations, we present the following inference rules which tell how one or more causations imply other causations. The inference rules refer to any set of terms $\mathbf{X}, \mathbf{Y}, \mathbf{Z} \in \mathbf{vars}(r)$.

Figure 4.1 Inference rules for causations

$$\begin{aligned}
&\text{If } \mathbf{Y} \subseteq \mathbf{X}, \text{ then } \mathbf{X} \rightsquigarrow \mathbf{Y}.\\
&\text{If } \mathbf{X} \rightsquigarrow \mathbf{Y}, \text{ then } \mathbf{X} \cup \mathbf{Z} \rightsquigarrow \mathbf{Y} \cup \mathbf{Z}.\\
&\text{If } \mathbf{X} \rightsquigarrow \mathbf{Y} \text{ and } \mathbf{Y} \rightsquigarrow \mathbf{Z}, \text{ then } \mathbf{X} \rightsquigarrow \mathbf{Z}.
\end{aligned} \tag{4.1}$$

The inference rules as presented in Figure 4.1 are almost an exact copy of the Armstrong's axioms [2] for functional dependencies. It can be shown, that they form a sound and a complete proof system for finiteness causations.

The notion of *closure*, which will be defined next, is useful when inferring causations of a rule. Informally, a closure of terms set $\mathbf{X}$ consists of all the $\mathbf{terms}(r)$ which are implied by $\mathbf{X}$.

Definition 4.2. *The* closure *of a set* $\boldsymbol{X} \subseteq \mathbf{terms}(r)$ *with respect to* $\mathcal{C}$ *and* r*, denoted* $\boldsymbol{X}^+_{\mathcal{C},r}$*, is the largest set* $\boldsymbol{Y} \subseteq \mathbf{terms}(r)$ *such that* $\mathcal{C} \models_r \boldsymbol{X} \rightsquigarrow \boldsymbol{Y}$*.*

For example, if $\mathcal{C} = \{\{\mathsf{child}_1\} \rightsquigarrow \{\mathsf{child}_2\}\}$, and the rule is given by τ_3 in the running example, then

$$\{\mathtt{y}\}^+_{\mathcal{C},\tau_3} = \{\mathtt{y}, \mathtt{y}', \mathtt{z}\}\,.$$

Lemma 4.1. *There exists a polynomial-time algorithm which, given a rule* r*, a set* $\mathcal{C}$*, and a set* $\boldsymbol{X} \subseteq \mathbf{vars}(r)$*, computes* $\boldsymbol{Y} = \boldsymbol{X}^+_{\mathcal{C},r}$*.*

Proof. Initially, $\mathbf{Y} \leftarrow \mathbf{X}$. For all $\sigma \in \mathcal{C}$ and all atoms $a \in \mathbf{body}(r)$, if $\sigma = \mathbf{u} \rightsquigarrow \mathbf{v}$ and $\mathsf{pos2term}_a(\mathbf{u}) \subseteq \mathbf{y}$, then add $\mathsf{pos2term}_a(\mathbf{v})$ to $\mathbf{y}$. Iterate until $\mathbf{Y}$ ceases to change. □

Algorithm 1 carries out the desired computation of closure.

Algorithm 1 $\mathsf{closure}(r, \mathcal{C}, \mathbf{X})$

Given a rule r, a constraints set $\mathcal{C}$, and a set $\mathbf{X} \subseteq \mathbf{terms}(r)$, return $\mathbf{X}^+_{\mathcal{C},r}$.

Let $\mathbf{X}^+_{\mathcal{C},r} \leftarrow \mathbf{X} \cup \mathbf{consts}(r)$
Repeat
  For all $1 \leq i \leq |\mathbf{body}(r)|$ **do** // *Fixed order iteration*
    Let a_i be the i^{th} atom of r
    If there exist $\mathbf{Y}, \mathbf{Z} \subseteq \mathbf{terms}(a_i)$ s.t.

(1) $\mathsf{term2pos}_{a_i}(\mathbf{Y}) \rightsquigarrow \mathsf{term2pos}_{a_i}(\mathbf{Z}) \in \mathcal{C}$,

(2) $\mathbf{Y} \subseteq \mathbf{X}^+_{\mathcal{C},r}$ and

(3) $\mathbf{Z} \nsubseteq \mathbf{X}^+_{\mathcal{C},r}$

    then
      $\mathbf{X}^+_{\mathcal{C},r} \leftarrow \mathbf{X}^+_{\mathcal{C},r} \cup \mathbf{Z}$ // $\boldsymbol{X}^+_{\mathcal{C},r}$ *implies also* $\boldsymbol{Z}$
    end If
  end for
until no more changes to $\mathbf{X}^+_{\mathcal{C},r}$
Return $\mathbf{X}^+_{\mathcal{C},r}$

The above algorithm runs in quadratic time in the size of the input ($\mathcal{C}$, $\mathbf{X}$ and $\mathbf{vars}(r)$). There are at most $|\mathbf{vars}(r)|$ iterations (each iteration increases $\mathbf{Y}$ by one argument in the worst case). In each iteration, all constraints in $\mathcal{C}$ are examined. There also exists a linear time algorithm for the closure computation [3].

The closure procedure is used in Algorithm 2 which computes the set $\mathcal{C}_r$ of constraints implied by $\mathcal{C}$ in r.

Algorithm 2 r_constraints(r, $\mathcal{C}$)

Return $\mathcal{C}_r$ for rule r and constraints set $\mathcal{C}$.

1: **Let** $\mathcal{C}_r \leftarrow \emptyset$
2: **Let H** $\leftarrow$ **terms**$(\mathrm{head}(r))$
3: **For all X** $\subseteq$ **H do** // *Find which variables are bound by* ***X***
4: **For all Y** $\subseteq \mathbf{X}^+_{\mathcal{C},r}$ **do** // *Adjust the result according to Definition 4.1*
5: $\mathcal{C}_r \leftarrow \mathcal{C}_r \cup \{\mathsf{term2pos}_r(\mathbf{X}) \rightsquigarrow \mathsf{term2pos}_r(\mathbf{Y})\}$
6: **end for**
7: **end for**
8: **Return** $\mathcal{C}_r$

The main loop of the algorithm, i.e., lines 3–5, is performed for all the subsets of variables appearing in the head term. For each such subset $\mathbf{X}$ the algorithm computes its closure $\mathbf{X}^+$. Now, since $\mathbf{X} \rightsquigarrow \mathbf{X}^+_{\mathcal{C},r}$ holds, it remains to translate this constraint (and subconstraints of it) to constraints over $\mathrm{pred}(r)$.

Lemma 4.2. *Algorithm 2 correctly computes* $\mathcal{C}_r$ *in time* $2^m\mathcal{O}(n^2)$*, where* $m = \mathrm{ar}(\mathrm{pred}(r)) + 1$ *and* n *is the length of* $\mathbf{vars}(r)$ *and* $\mathcal{C}$*.*

Proof. Omitted. □

Chapter 5

Program Wide Constraints Implication

Now that the means for inferring constraints within a single rule are established, we are ready to study the more interesting problem, i.e., inference of constraints with respect to an entire DATALOG program Π. In doing so, we will need to take into account the effects of multiple applications of the same rule, the fact that an IDB may be defined by more than one rule, and that the definition of different IDBs may be mutually recursive.

In this section let $\mathcal{C}$ be a fixed set of constraints on the extensional predicates of Π and let $\mathcal{D}$ be a database, i.e., a set of p-facts, where $p \in \mathbf{edb}(\Pi)$.

Definition 5.1. *Let $p \in \mathbf{idb}(\Pi)$ and let σ be a constraint on p. We say that $\mathcal{C}$* implies *σ, denoted $\mathcal{C} \models \sigma$, if the set $\Pi_p^n(\mathcal{D})$ satisfies the constraint σ for all $n \geq 0$ whenever $\mathcal{D} \models \mathcal{C}$.*

Remark 5.2. *The implication considered in this research is* finite implication*, i.e., a constraint is implied if it holds in all finite number of rule applications. Deciding which constraints hold after infinitely many applications, allows one to decide termination, and is therefore undecidable.*

Henceforth, we shall assume that $\mathcal{D}$ satisfies $\mathcal{C}$. Let $\mathcal{C}_p$ be the set of all the implied constraints over $p \in \mathbf{idb}(\Pi)$, i.e., all constraints $\mathbf{x} \rightsquigarrow \mathbf{y}$, where $\mathbf{x}, \mathbf{y} \subseteq \mathbf{pos}(p)$ and $\mathcal{C} \models \mathbf{x} \rightsquigarrow \mathbf{y}$. Let $\mathcal{C}_\Pi$ denote the set of all the constraints on IDB predicates of Π.

Observe that if no facts are established for a certain predicate p, i.e., no p-facts exist in $\mathcal{F}$, then $\mathcal{F}$ satisfies *any* constraint $\sigma = \mathbf{x} \rightsquigarrow \mathbf{y}$, where $\mathbf{x}, \mathbf{y} \subseteq \mathbf{pos}(p)$, This is precisely the circumstances for all $p \in \mathbf{idb}(\Pi)$, when program Π starts. The set $\mathcal{F}$ will continue to satisfy σ if no rule defining p will ever generate facts that violate σ.

These observations are employed in Algorithm 3 which uses a fixed point evaluation strategy for computing $\mathcal{C}_\Pi$. Algorithm 3 maintains the set $\mathcal{P}_p$ of constraints for every

Algorithm 3 program_constraints$(\Pi, \mathcal{C})$

Given a program Π, and a set $\mathcal{C}$ of constraints over its extensional predicates, return $\mathcal{C}_\Pi$.

1: **For all** $p \in \mathbf{idb}(\Pi)$ **do** // *find IDB candidate constraints*
2: **Let** $\mathcal{P}_p \leftarrow \{\mathbf{x} \rightsquigarrow \mathbf{y} \mid \mathbf{x}, \mathbf{y} \subseteq \mathbf{pos}(p)\}$
3: **end for**
4: **let** $\mathcal{C}_\Pi \leftarrow \mathcal{C} \cup \bigcup_{p \in \mathbf{idb}(\Pi)} \mathcal{P}_p$ // *add candidates to* $\mathcal{C}_\Pi$
5: **Repeat** // *Invalidate constraints until each* $\mathcal{P}_p$ *is reduced to* $\mathcal{C}_p$
6: **For all** $p \in \mathbf{idb}(\Pi)$ **do** // *refine* $\mathcal{P}_p$ *as implied by* $\mathcal{C}_\Pi$
7: **For all** $\alpha \in \Pi, \mathrm{pred}(\alpha) = p$ **do** // *examine all* p*-rules*
8: **let** $\mathcal{C}_\alpha \leftarrow$ r_constraints$(\alpha, \mathcal{C}_\Pi)$
9: $\mathcal{C}_\Pi \leftarrow \mathcal{C}_\Pi \setminus \mathcal{P}_p$ // *forget all* p*-constraints regarding* p
10: $\mathcal{P}_p \leftarrow \mathcal{P}_p \cap \mathcal{C}_\alpha$ // *remove constraints not preserved by* α
11: $\mathcal{C}_\Pi \leftarrow \mathcal{C}_\Pi \cup \mathcal{P}_p$ // *revive* p*-constraints preserved by* α
12: **end for**
13: **end for**
14: **until** no changes in $\mathcal{C}_\Pi$
15: **Return** $\mathcal{C}_\Pi$

intensional predicate $p \in \Pi$. Initially, the algorithm assumes that all the constraints are satisfied by the set $\mathcal{F}$ of p-facts (line 2). Then, the algorithm iteratively eliminates the constraints which are definitely not satisfied by $\mathcal{F}$ until a fixed point is reached. In particular, a constraint σ is in $\mathcal{C}_p$, if σ is implied (in a steady state) by all the rules defining p.

Before moving on towards a correctness proof, we would like to demonstrate the run of the algorithm on a concrete example. Consider the following program which computes Bill's ancestors in its goal predicate.

Example 5.3.

$$\begin{aligned} r_1&\colon \mathsf{q}(\mathtt{y}) \leftarrow \mathsf{heir}(\textit{`Bill'}, \mathtt{y}).\\ r_2&\colon \mathsf{heir}(\mathtt{x}, \mathtt{y}) \leftarrow \mathsf{child}(\mathtt{x}, \mathtt{z}), \mathsf{heir}(\mathtt{z}, \mathtt{y}).\\ r_3&\colon \mathsf{heir}(\mathtt{x}, \mathtt{y}) \leftarrow \mathsf{child}(\mathtt{x}, \mathtt{y}). \end{aligned} \tag{5.1}$$

Here the heir *predicate is defined in the same way as in our running example (see Figure 2.1). Assuming that* $\{\mathsf{child}_1\} \rightsquigarrow \{\mathsf{child}_2\}$, *the algorithm will infer constraints over* heir *and* q *predicates.*

At its initialization (line 2) the algorithm creates two sets:

$$\mathcal{P}_{\mathsf{q}} = \{\emptyset \rightsquigarrow \{\mathsf{q}_1\}\}\,, \tag{5.2}$$

and

$$\begin{aligned}\mathcal{P}_{\mathsf{heir}} = \{&\emptyset \rightsquigarrow \{\mathsf{heir}_1\}\,, \emptyset \rightsquigarrow \{\mathsf{heir}_2\}\,,\\ &\{\mathsf{heir}_1\} \rightsquigarrow \{\mathsf{heir}_2\}\,,\\ &\{\mathsf{heir}_2\} \rightsquigarrow \{\mathsf{heir}_1\}\}\end{aligned} \tag{5.3}$$

Let us assume that in the main loop (lines 5–14) the algorithm first handles the predicate q *and then the predicate* heir*. Also, assume that the recursive* heir*-rule is examined first (line 7). Then, Algorithm 3 updates the sets $\mathcal{P}_{\mathsf{q}}$ and $\mathcal{P}_{\mathsf{heir}}$ as follows:*

1. The 1^{st} examination of rule r_1 *remains the set $\mathcal{P}_{\mathsf{q}}$ unchanged.*
2. The 1^{st} examination of rule r_2 *reduces the set $\mathcal{P}_{\mathsf{heir}}$ by two constraints:* $\emptyset \rightsquigarrow \{\mathsf{heir}_1\}$ *and* $\{\mathsf{heir}_2\} \rightsquigarrow \{\mathsf{heir}_1\}$.
3. The 1^{st} examination of rule r_3 *also eliminates* $\emptyset \rightsquigarrow \{\mathsf{heir}_2\}$ *from* $\mathcal{P}_{\mathsf{heir}}$.
   At this stage the set contains only one constraint, i.e.,

   $$\mathcal{P}_{\mathsf{heir}} = \{\{\mathsf{heir}_1\} \rightsquigarrow \{\mathsf{heir}_2\}\}\,.$$

4. *Any subsequent examinations of the program's rules remain the constraints sets unchanged.*

Thus, the output of the algorithm is

$$\begin{aligned}&\mathcal{P}_{\mathsf{q}} = \{\emptyset \rightsquigarrow \{\mathsf{q}_1\}\}\,,\\ &\mathcal{P}_{\mathsf{heir}} = \{\{\mathsf{heir}_1\} \rightsquigarrow \{\mathsf{heir}_2\}\}\,.\end{aligned}$$

A bit of notation is required in reasoning about the algorithm. We assume that the algorithm uses a fixed order for iterating over predicates and rules. With this order, there is a global enumeration of the iterations of the *inner loop* (lines 9–11).

An i^{th} superscript attached to a variable used by the algorithm denotes the value of this variable at the beginning of the i^{th} iteration. For example, $\mathcal{P}_p^3$ is the value of $\mathcal{P}_p$ at the beginning of the 3^{rd} execution of the inner loop, i.e., the value which is assigned to $\mathcal{P}_p$ at the 2^{nd} execution of line 10. With this notation we have,

$$\mathcal{C}_\Pi^i = \mathcal{C} \cup \bigcup_{p \in \mathbf{idb}(\Pi)} \mathcal{P}_p^i \tag{5.4}$$

for all $i > 0$, while the main computation carried out by the algorithm, i.e., line 10, can be written as the following recursion

$$\mathcal{P}_p^{i+1} = \begin{cases} \mathcal{P}_p^i \cap \mathsf{r\_constraints}(\alpha^i, \mathcal{C}_\Pi^i) & \text{if } \mathrm{pred}(\alpha^i) = p, \\ \\ \mathcal{P}_p^i & \text{otherwise.} \end{cases} \tag{5.5}$$

for all $p \in \mathbf{idb}(p)$ and all $i > 0$.

An $*$ superscript will denote the value of a variable at the end of the last iteration (for now just assume that the algorithm always terminates). It is mundane to check that the sequence of approximations to every individual predicate are non-increasing, i.e.,

$$\mathcal{P}_p^1 \supseteq \mathcal{P}_p^2 \supseteq \cdots \supseteq \mathcal{P}_p^*. \tag{5.6}$$

Also, the sequence of approximations to the collective set of constraints is non-increasing, i.e.,

$$\mathcal{C}_\Pi^1 \supseteq \mathcal{C}_\Pi^2 \supseteq \cdots \supseteq \mathcal{C}_\Pi^m \supseteq \mathcal{C}_\Pi^{m+1} = \mathcal{C}_\Pi^*, \tag{5.7}$$

where m is the total number of iterations. (It is convenient to assume that there is an empty dummy iteration which takes place after the last iteration, in which no variables are changed.)

Examining the computation carried out at iteration i we determine that if a set of facts $\mathcal{F}$ satisfies $\mathcal{C}_\Pi^i$, then an application of rule α^i to $\mathcal{F}$ results in a set of facts which satisfies $\mathcal{C}_\Pi^{i+1}$, and in particular $\mathcal{P}_p^{i+1}$. Formally,

$$\mathcal{F} \models \mathcal{C}_\Pi^i \;\Rightarrow\; \alpha_p^i(\mathcal{F}) \models \mathcal{P}_p^{i+1}. \tag{5.8}$$

Also, it follows from the termination condition (line 14) that no rule application can generate facts that violate $\mathcal{C}^*_\Pi$, i.e,

$$\mathcal{F} \models \mathcal{C}^*_\Pi \Rightarrow r(\mathcal{F}) \models \mathcal{C}^*_\Pi. \tag{5.9}$$

for all $r \in \Pi$.

Lemma 5.1. *Set $\mathcal{C}^*_\Pi$ bounds below the constraints in any run of Π, i.e., $\mathcal{C}^*_\Pi \subseteq \mathfrak{C}(\vec{\boldsymbol{r}}(\mathcal{D}))$ for every sequence $\vec{\boldsymbol{r}}$ with rules drawn from Π.*

Proof. By simultaneous induction on $n = |\vec{\boldsymbol{r}}|$, noting that

$$\mathcal{C}^*_\Pi = \mathcal{C} \cup \bigcup_{p \in \mathbf{idb}(\Pi)} \mathcal{P}^*_p.$$

If $n = 0$, then $\vec{\boldsymbol{r}}(\mathcal{D}) = \mathcal{D}$ and by assumption $\mathcal{D}$ satisfies $\mathcal{C}$. Also, all sets $\vec{\boldsymbol{r}}_p(\mathcal{D})$ are empty, and hence satisfy all possible constraints and in particular $\mathcal{P}^*_p$.

Let $n > 0$, then let $\vec{\boldsymbol{r}}'$ be a sequence and r be a rule such that $\vec{\boldsymbol{r}} = \vec{\boldsymbol{r}}' r$. By the inductive hypothesis,

$$\mathcal{C}^*_\Pi \subseteq \mathfrak{C}(\vec{\boldsymbol{r}}'(\mathcal{D})),$$

i.e., $\vec{\boldsymbol{r}}'(\mathcal{D}) \models \mathcal{C}^*_\Pi$. Using (5.9) with $\vec{\boldsymbol{r}}'(\mathcal{D})$ we obtain

$$r(\vec{\boldsymbol{r}}'(\mathcal{D})) \models \mathcal{C}^*_\Pi,$$

i.e.,

$$\mathcal{C}^*_\Pi \subseteq \mathfrak{C}(\vec{\boldsymbol{r}}(\mathcal{D})).$$

□

The lemma also implies that $\mathcal{P}^*_p \subseteq \mathfrak{C}(\vec{\boldsymbol{r}}_p(\mathcal{D}))$, for all $p \in \mathbf{idb}(\Pi)$ and for every sequence $\vec{\boldsymbol{r}}$ of Π's rules.

We now consider the particular sequence of rules that the algorithm selects in the course of its run. Let $\vec{\boldsymbol{\alpha}}$ be a prefix of the sequence of α^i's, i.e.,

$$\vec{\boldsymbol{\alpha}} = \alpha^1 \cdots \alpha^k \text{ for some } k \geq 0.$$

Lemma 5.2. *For all $p \in \mathbf{idb}(\Pi)$, it holds that*

$$\mathfrak{C}(\vec{\boldsymbol{\alpha}}_p(\mathcal{D})) = \mathcal{P}^{k+1}_p.$$

Proof. By induction on k for all $p \in \mathbf{idb}(\Pi)$. The inductive base holds since if $|\vec{\boldsymbol{\alpha}}| = 0$, then $\vec{\boldsymbol{\alpha}}_p(\mathcal{D}) = \emptyset$, and $\mathcal{P}_p^1$ is the set of all constraints.

Consider the case that $k > 0$. Let $\vec{\boldsymbol{\alpha}}' = \alpha^1 \cdots \alpha^{k-1}$, i.e., $\vec{\boldsymbol{\alpha}} = \vec{\boldsymbol{\alpha}}'\alpha^k$. By the inductive hypothesis,

$$\mathfrak{C}(\vec{\boldsymbol{\alpha}}'_p(\mathcal{D})) = \mathcal{P}_p^k. \tag{5.10}$$

If α^k does not define p then

$$\vec{\boldsymbol{\alpha}}'_p(\mathcal{F}) = \vec{\boldsymbol{\alpha}}_p(\mathcal{F})$$

for all $\mathcal{F}$. It follows from (5.5) that $\mathcal{P}_p^{k+1} = \mathcal{P}_p^k$, and the induction step holds.

Otherwise, let Δ be the set of p-tuples which the application of α^k produced, i.e.,

$$\vec{\boldsymbol{\alpha}}_p(\mathcal{D}) = \vec{\boldsymbol{\alpha}}'_p(\mathcal{D}) \cup \Delta, \tag{5.11}$$

Applying Fac. 3.5 we obtain

$$\mathfrak{C}(\vec{\boldsymbol{\alpha}}_p(\mathcal{D})) = \mathfrak{C}(\vec{\boldsymbol{\alpha}}'_p(\mathcal{D})) \cap \mathfrak{C}(\Delta). \tag{5.12}$$

Lemma 4.2 establishes

$$\mathfrak{C}(\Delta) = \mathsf{r\_constraints}(\alpha^k, \mathcal{C}_\Pi^k).$$

The lemma now follows by applying again (5.5) and using the inductive hypothesis (5.10). □

Theorem 5.1. *There exists an exponential time algorithm which computes* $\mathcal{C}_\Pi$.

Proof. Lemma 5.1 implies that Algorithm 3's return value, $\mathcal{C}_\Pi^*$, does not contain invalid constraints, i.e., $\mathcal{C}_\Pi^* \subseteq \mathcal{C}_\Pi$. To show that there is no valid constraint which $\mathcal{C}_\Pi^*$ does not contain, apply Lemma 5.2 in the case $|\vec{\boldsymbol{\alpha}}| = m$, i.e., the longest such $\vec{\boldsymbol{\alpha}}$. From (5.4) it follows that

$$\mathfrak{C}(\vec{\boldsymbol{\alpha}}(\mathcal{D})) = \mathcal{C}_\Pi^{m+1}$$

which, using (5.7) implies

$$\mathfrak{C}(\vec{\boldsymbol{\alpha}}(\mathcal{D})) = \mathcal{C}_\Pi^*.$$

It follows that $\mathcal{C}^*_\Pi \supseteq \mathcal{C}_\Pi$.

To see that the algorithm terminates, use (5.7) while noticing that $\mathcal{C}^1_\Pi$ is finite, and $\mathcal{C}^i_\Pi$ must decrease after every m iterations, where m is the number of rules in Π. The runtime of the algorithm is exponential in the worst case, since the set of all possible constraints of predicate p, i.e., the value computed at 4, is exponential in $\mathrm{ar}(p)$. Thus, $\mathcal{C}^1_\Pi$ may be exponentially sized, and it could be the case that each iteration eliminates only a polynomial number of constraints. □

Note that $\mathcal{C}_\Pi$ as received by Algorithm 3 may contain redundant constraints. For example, if $\{p_1\} \rightsquigarrow \{p_1, p_2, p_3\} \in \mathcal{C}_\Pi$ then $\mathcal{C}_\Pi$ also contains $\{p_1\} \rightsquigarrow \{p_1\}$ and $\{p_1\} \rightsquigarrow \{p_2\}$ and etc. Precisely, there are $2^3 = 8$ constrains which represent what is constrained by $\{p_1\}$. This is the result of Algorithm 2 (see line 5).

Sometimes we would like to work with the *minimized* version of $\mathcal{C}_\Pi$, denoted $\mathsf{min}(\mathcal{C}_\Pi)$, in which no redundant constraints of this kind present. In particular, $\mathsf{min}(\mathcal{C}_\Pi)$ holds that if $\mathbf{x} \rightsquigarrow \mathbf{y} \in \mathsf{min}(\mathcal{C}_\Pi)$, then there are no constraints $\mathbf{x} \rightsquigarrow \mathbf{z} \in \mathsf{min}(\mathcal{C}_\Pi)$ such that $\mathbf{z} \subseteq \mathbf{y}$.

Chapter 6

Deciding Weak Safety

In this section we present the theorem which decides the weak safety problem.

Theorem 6.1. *Let* Π *be a* DATALOG *program, and let* q *be its goal. Then,* Π *is* weakly safe *iff* $\mathcal{C} \models \emptyset \rightsquigarrow \mathbf{pos}(q)$.

Proof. If Π is weakly safe, then according to Definition 3.7,

$$\Pi_q^n(\mathcal{D})$$

is finite for all $n \geq 0$ whenever $\mathcal{D} \models \mathcal{C}$. The above is possible only if $\mathcal{C} \models \emptyset \rightsquigarrow \mathbf{pos}(q)$.

Conversely, assume that $\mathcal{C} \models \emptyset \rightsquigarrow \mathbf{pos}(q)$. Then, by Definition 5.1, the set

$$\mathcal{F}_n = \Pi_q^n(\mathcal{D})$$

satisfies $\emptyset \rightsquigarrow \mathbf{pos}(q)$ for all $n \geq 0$ whenever $\mathcal{D} \models \mathcal{C}$. Finally, according to Definition 3.2 it follows that the set

$$\{a \mid a \in \mathcal{F}_n \wedge \mathrm{pred}(a) = q\}$$

is finite for all $n \geq 0$. The claim follows. □

Consider, for example, the following DATALOG program, which computes ancestors of 'Rachel':

$$\begin{aligned} &\mathsf{heir}(\mathtt{x}, \mathtt{y}) \leftarrow \mathsf{child}(\mathtt{x}, \mathtt{y}). \\ &\mathsf{heir}(\mathtt{x}, \mathtt{y}) \leftarrow \mathsf{child}(\mathtt{x}, \mathtt{z}), \mathsf{heir}(\mathtt{z}, \mathtt{y}). \\ &\mathsf{rachel\_ancestor}(\mathtt{x}) \leftarrow \mathsf{heir}(\text{'Rachel'}, \mathtt{x}). \end{aligned} \tag{6.1}$$

Suppose that the input database of program (6.1) satisfies $\mathcal{C} = \{\{\mathsf{child}_1\} \rightsquigarrow \{\mathsf{child}_2\}\}$. Algorithm 3 deduces that

$$\{\mathsf{heir}_1\} \rightsquigarrow \{\mathsf{heir}_2\}$$

holds and so is $\emptyset \rightsquigarrow \{\mathsf{rachel\_ancestor}_1\}$. The last constraint implies that any finite number of rule applications deduces finitely many rachel_ancestor-facts. This observation makes program (6.1) weakly safe.

Remark 6.1. *Theorem 6.1 establishes that $\mathcal{C}_\Pi$ can be used to decide weak safety. But, since weak safety* EXP*-time complete [23], it is no wonder that our algorithm for computing $\mathcal{C}_\Pi$ is exponential.*

Chapter 7

Deciding Termination

If a program is weakly safe, then any finite number of rule applications contributes a finite number of facts to the program semantics. However, the result of a weakly safe program may include an unbounded number of facts, since in general, the number of rule applications is unbounded. Indeed, Sagiv and Vardi [23] showed that the independent problem of termination is undecidable, without being able to produce an algorithm for determining termination in the case that weak safety is known, or conversely, to prove that no such algorithm exists.

This section sets conditions, common in tasks of processing software, which exclude the situation that a program is weakly safe yet not terminating.

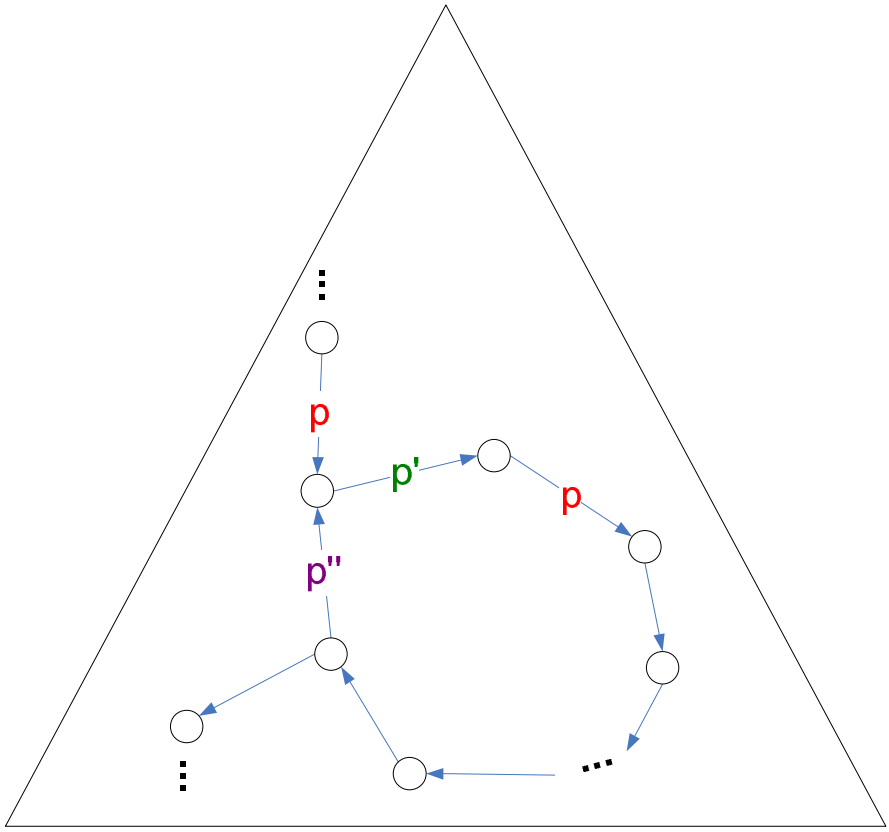
Specifically, we show that every weakly safe program is also terminating whenever the database is *founded*.

We start from informal definition of *founded database* which is quite intuitive. Founded databases are restricted to contain only binary EDB predicates. Therefore it is natural to represent a database as an infinite graph, in which edges correspond to facts. For each predicate p having the constraint $\{\mathsf{p}_1\} \rightsquigarrow \{\mathsf{p}_2\}$, an edge $c_1 \rightarrow c_2$ is created for each fact $\mathsf{p}(c_1, c_2)$. Similarly, for each predicate p having the constraint $\{\mathsf{p}_2\} \rightsquigarrow \{\mathsf{p}_1\}$, an edge $c_2 \rightarrow c_1$ is created for each fact $\mathsf{p}(c_1, c_2)$. We say that a database is founded if every infinite path contains only finitely many vertices.

Consider Figure 7.1 which schematically represents database as an infinite graph. Here the database contains at least three EDB predicates: p, p' and p". Edges marked with a predicate name represent facts of that predicate. In this figure one can see an infinite path. If the represented database is founded, then the path traverses only finitely

many vertices.

Figure 7.1 Database as an infinite graph.



Now we are ready to define precisely what founded database means.

Definition 7.1. *A database $\mathcal{D}$ satisfying a set of constraints $\mathcal{C}$ is* founded *if:*

1. *all EDB predicates are binary*
2. *there are only finitely many distinct elements in every infinite sequence $\ell_1, \ell_2, \ldots$ in which every consecutive pair ℓ_i, ℓ_{i+1}, $i \geq 1$ satisfies at least one of the following:*
   (a) *$p(\ell_i, \ell_{i+1})$ holds for some EDB predicate p and $\{p_1\} \rightsquigarrow \{p_2\} \in \mathcal{C}$*
   (b) *$p(\ell_{i+1}, \ell_i)$ holds for some EDB predicate p and $\{p_2\} \rightsquigarrow \{p_1\} \in \mathcal{C}$.*

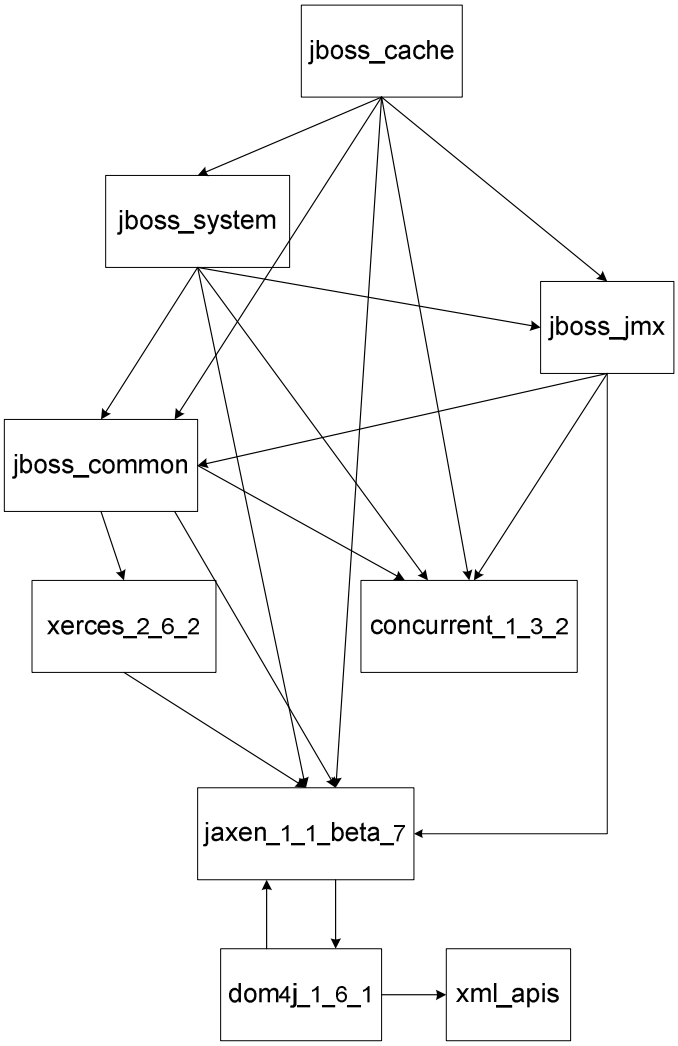
Consider the database which represents relations between programming units. Such database contains relations such as “inherits”, “calls”, “depends” etc. It is obvious to see

that this database is founded, since the number of units used by a certain programming module must be finite (otherwise the compilation process will never end).

For a more concrete example consider JAVA jar files which may be dependent one upon the other, sometimes even in a cyclic way. This kind of dependencies can be found in Hibernate [1] open-source project. Hibernate is an object-relational mapping library for the JAVA language, which provides a framework for mapping an object-oriented domain model. It contains 36 jar files with over 70 dependencies between them.

Figure 7.2 partially illustrates dependencies between Hibernate jar files.

Figure 7.2 Hibernate jar files dependencies.



Here, a cyclic dependency exists between two jar files: jaxen_1_1_beta_7 and

dom4j_1_6_1. Therefore, infinite sequences of jar files are possible. Nevertheless, all infinite sequences must contain finitely many elements due to reasons explained before.

Remark 7.2. *One may mistakenly think that for an infinite sequences to contain finitely many elements it must recurse indefinitely in a cycle. This is not the case. Consider the transcendental number* e*, the base of the natural logarithm. Since* e *is irrational number, its decimal expansion never terminates nor repeats itself. Yet', it contains at most 10 distinct digits.*

Now we are ready to state the central theorem of this section.

Theorem 7.1. *Let* $\mathcal{D}$ *be a founded database satisfying the set of constraints* $\mathcal{C}$*. Then, if* Π *is weakly safe with respect to* $\mathcal{C}$*, then it is also terminating over* $\mathcal{D}$*.*

The theorem can be made a bit more general, dealing with unary EDB predicates. To simplify the presentation, we omit this generalization.

Henceforth assume that Π is indeed weakly safe with regards to $\mathcal{C}$ and $\mathcal{D}$ is founded. To prove the theorem we first write the yield of q-facts of every possible sequence of rule applications as a set of expansion rules defining q (as in (2.3)).

In the running example, an expansion rule that corresponds to the shortest sequence of rule applications that may generate a q-fact is obtained by adding two atoms to the body of expansion rule (2.2):

$$\begin{aligned} \mathsf{q}(\mathtt{x},\mathtt{y}) \leftarrow\ & \mathsf{child}(\mathtt{x},\mathtt{u}), \mathsf{child}(\mathtt{x},\mathtt{v}), \mathsf{not\_eq}(\mathtt{u},\mathtt{v}), \mathsf{child}(\mathtt{u}',\mathtt{y}), \\ & \mathsf{child}(\mathtt{u},\mathtt{u}'), \mathsf{child}(\mathtt{v}',\mathtt{y}), \mathsf{child}(\mathtt{v},\mathtt{v}'), \\ & \mathsf{dependant}(\mathtt{x},\mathtt{y}), \mathsf{child}(\text{`Bill'},\mathtt{x}), \mathsf{child}(\mathtt{w},\text{`Bill'}). \end{aligned} \tag{7.1}$$

Fix an enumeration of these rules, $\gamma^1, \gamma^2, \ldots$. The proof is carried out by showing that the set

$$\bigcup_{i=1}^{\infty} \left(\gamma^i(\mathcal{D}) \setminus \mathcal{D}\right)$$

is finite. We show in fact that there is a finite set of representative rules (which are not necessarily expansion rules)

$$\gamma^{i_1}, \ldots, \gamma^{i_k},$$

such that for every expansion rule γ^i there is a representative rule γ^j, where $j \in \{i_1, \ldots, i_k\}$ such that

$$\gamma^i(\mathcal{D}) \subseteq \gamma^j(\mathcal{D}).$$

Theorem 7.1 will follow from the observation that each such

$$\gamma^j(\mathcal{D}) \setminus \mathcal{D}$$

is finite when Π is weakly safe.

7.1 Expansion Graphs

Let γ and γ' be two expansion rules which define q. When can we be certain that $\gamma(\mathcal{D}) = \gamma'(\mathcal{D})$? This is obviously the case when γ can be obtained from γ' by reordering body atoms and renaming of temporary variables, i.e., variables not occurring in the head. To capture this equivalence we represent each expansion rule γ^i as an edge- and vertex-colored graph G_i. We shall argue that rules γ^i and γ^j generate the same set of q-facts if their graphs are isomorphic (differ only in names of vertices with the same color).

The intuition behind representative rules creation goes one small step further. We argue that the additions of vertices or edges to G_i cannot increase $\gamma^i(\mathcal{D})$. Also, in the case that $\mathcal{D}$ is founded, there is a limited variety of such graphs. The set $\gamma^{i_1}, \ldots, \gamma^{i_k}$ is created so that one of $G_{i_1}, \ldots, G_{i_k}$ is a subgraph of G_i for an arbitrary i.

The graph representation of an expansion rule γ is obtained by making an edge for each atom a which occurs in the body of γ. (This is possible since by definition $\mathrm{pred}(a) \in \mathbf{edb}(\Pi)$ and by assumption $\mathrm{ar}(a) = 2$.) The graph vertices are precisely **terms**(γ). We partition this set into three disjoint subsets:

1. *sources*, defined by

$$\textbf{sources}(\gamma) = \{t \mid t \in \textbf{terms}(\gamma) \wedge \mathcal{C} \models_\gamma \emptyset \rightsquigarrow \{t\}\},$$

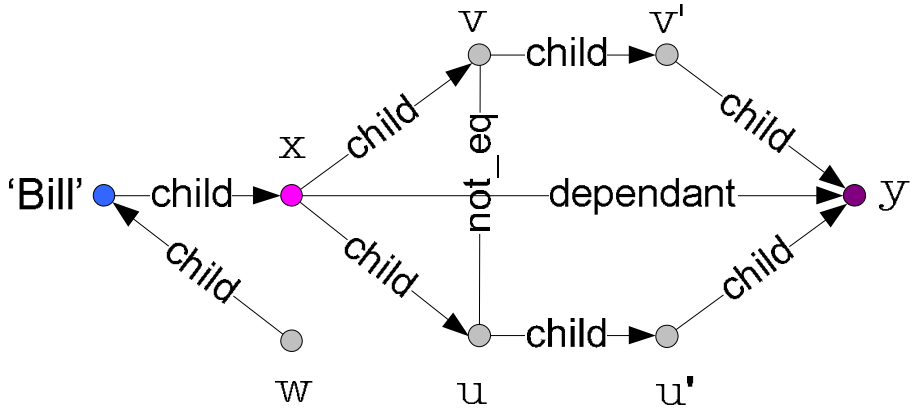
2. *targets*, given by the set **targets**(γ), which are variables occurring in the head of γ, and

3. *temporaries*, which are all the remaining variables, denoted by **free**(γ)

Note that if a program Π is weakly safe, then there exists at least one source in the expansion graph. However it can be the case that there are no targets in the graph. This occurs when the goal predicate is boolean (i.e., contains no terms in its head).

The graph representation of expansion rule (7.1) is depicted in Figure 7.3. In this example, the only source is 'Bill', targets are x and y, while the temporaries are u, v, u′, v′ and w.

Figure 7.3 The expansion graph of expansion rule (7.1).



The formal definition of this graph representation is as follows.

Definition 7.3. *Let γ be an expansion rule. Then the* expansion graph *of γ has* **terms**(γ) *as vertices. Each source and each target is assigned a unique color, whereas the temporaries are colored with a common neutral color. For each atom $p(v, v')$ in the body of γ, this graph has a p-colored (v, v').*

We see that in Figure 7.3, there are a total of 10 edges, one for each body atom in (7.1). The graph uses three different colors (i.e., labels) for edges, one for each EDB predicate. Also, there are four colors for vertices (i.e., for 'Bill', x, y and the temporary variables).[1]

Clearly, any program Π sets a finite palette of colors that any graph G may use. We will study the family of such graphs that Π may generate.

Definition 7.4. *An expansion graph $G = (V, E)$ is* isomorphic *to an expansion graph $G' = (V', E')$, if there is a bijective function*

$$f : V \to V',$$

[1] In a printout of this work, the colors of the vertices may not be apparent.

such that $f(v) = v$ if v is a source or a target, and if $\langle u, v\rangle$ is an edge in E with color p, then $\langle f(u), f(v)\rangle$ is an edge in E' with color p.

It should be obvious that an expansion rule can be constructed from every expansion graph. Also, if two graphs are isomorphic, then the rules obtained from them produce the same set of q-facts.

Definition 7.5. *Graph G is a* subgraph *of a graph G' if G is isomorphic to a graph obtained from G' by removing any number of edges and vertices.*

Lemma 7.1. *Let γ and γ' be expansion rules, and let G and G' be (respectively) their expansion graphs. Then, if G is a subgraph of G', then*

$$\gamma'(\mathcal{D}) \subseteq \gamma(\mathcal{D}).$$

Proof. The atoms in the body of γ make a subset of the atoms of the body of γ'. □

7.2 Directionality in Expansion Graphs

A p-colored edge (v, v') in an expansion graph represents an atom $p(v, v')$ in the rule. To represent EDB constraints, we shall assign directionality with edges: if

$$\{p_1\} \rightsquigarrow \{p_2\}$$

holds, then edge (v, v') is *co-directed*, i.e., its direction is from v to v'. But, if

$$\{p_2\} \rightsquigarrow \{p_1\}$$

holds then (v, v') is *contra-directed*, i.e., its direction is from v' to v, opposing the edge syntax. The edge will be *bidirectional* if p has both constraints, and will be *undirected* if p has no constraints associated with it.

Arrows are used in Figure 7.3 to denote edge directions. In this example, there is only one undirected edge, and no bidirectional or contra-directed edges. All the other edges are *co-directed*.

Edge directions in the graph of an expansion rule γ, represent causations of γ due to EDB constraints. Moreover, the implied causations of γ are obtained by a simple

transitive closure of the directionality relation: a *directed* path is a path in an expansion graph which traverses edge only according to their directionality. Thus, directed paths cannot traverse undirected edges.

We use directed paths to infer new causations. In fact, examining Algorithm 2 we obtain,

Lemma 7.2. *Let γ be an expansion rule of Π, and let G be its expansion graph. Then,*

$$\mathcal{C} \models_\gamma \{\mathtt{x}\} \rightsquigarrow \{\mathtt{y}\}$$

if and only if there exists in G a directed path from node x *to node* y.

The set of constraints on q that an expansion rule γ imply may be quite general. However, for weakly safe programs, there is a common core to all these sets:

Lemma 7.3. *Let γ be an expansion rule of a weakly safe program Π which defines q. Then,*

$$\mathcal{C} \models_\gamma \emptyset \rightsquigarrow \mathbf{pos}(q).$$

Proof. Follows from Theorem 6.1 and Definition 5.1 and the fact that γ equals to a finite sequence of Π's rules. □

Combining the above two lemmas we obtain:

Corollary 7.6. *Let γ be an expansion rule of Π and let G be its expansion graph. Then, for every target $t \in$ **targets**(γ) there is a source $s \in$ **sources**(γ) such that there exists a directed path in G leading from s to t.*

Let a *computational path* be as in the above corollary, i.e., a directed path leading from a source to a target. Figure 7.3 for example, has exactly four computational paths, all starting at vertex 'Bill'; one of these ends in vertex x while the other three end in vertex y.

7.3 Reduced Expansion Graphs

To finish the proof of Theorem 7.1, we shall use the directionality of expansion graphs, together with the foundedness property of the domain, to restrict the number of possible

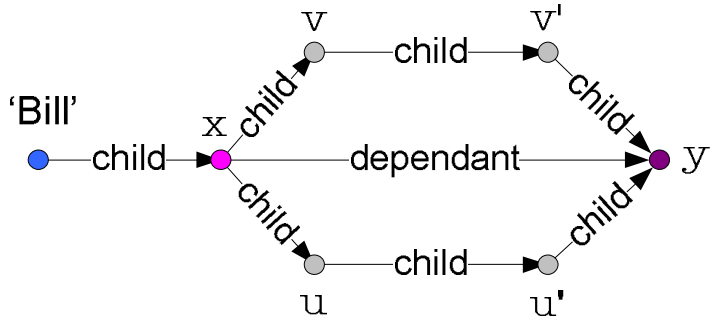
such graphs. Corollary 7.6 established that each target is defined by at least one computational path. We shall argue that in the conditions of the theorem: (i) the number of sources and targets is bounded; (ii) the number of distinct values, which can be assigned to every internal nodes of any computational path is bounded; and (iii) edges which do not take part in computational paths are immaterial in a sense.

Claim (i) is obvious, since any given DATALOG program has a bounded number of constants. Also, there are finitely many sources due to predicates having constraints of the form $\emptyset \rightsquigarrow \mathbf{x}$. Claim (ii) is the subject of Section 7.4. Claim (iii) can be stated more formally with the following definition

Definition 7.7. *Let G be an expansion graph of γ. The* reduced *graph of G, denoted $\widetilde{G}$, is a graph obtained from G by removing all edges which do not appear in any computational path.*

The reduced expansion graph of expansion rule (7.1) is depicted in Figure 7.4. Observe that the not_eq-edge has been removed, as well as the incoming child-edge to the node 'Bill'.

Figure 7.4 The reduced expansion graph of expansion rule (7.1).



Clearly, every reduced graph $\widetilde{G}$ defines a q-rule (which is not necessarily an expansion rule of Π). Since the reduction process preserves all computational paths, and these paths go along EDB constraints, then the number of q-facts that this rule produces is finite.

Lemma 7.4. *Let γ be an expansion rule, G be its expansion graph, and $\widetilde{\gamma}$ be the rule of $\widetilde{G}$. Then $\widetilde{\gamma}_q(\mathcal{D})$ is finite, and*

$$\gamma(\mathcal{D}) \subseteq \widetilde{\gamma}(\mathcal{D})$$

for any $\mathcal{D}$.

Proof. Follows from Lemma 7.1 and Corollary 7.6. □

7.4 Bounded Repertoire of Reduced Expansion Graphs

Lemma 7.4 established the process by which the selection of representatives is carried out: Start from all q-rules in

$$\gamma^1, \gamma^2, \ldots .$$

For each such rule, build its expansion graph G_i, and then its reduced expansion graph $\widetilde{G_i}$. The representatives $\gamma^{i_1}, \ldots, \gamma^{i_k}$ are nothing but the rules of the graphs $\widetilde{G_i}$.

We cannot show that the number of distinct reduced expansion graphs is finite. Instead we show that there is a bound beyond which expansion graphs cannot yield new q-facts. To do so, recall how a reduced graph (actually the rule which it defines) is used to deduce q-facts. Every such fact is produced by a *consistent* assignment of values drawn from $\mathbb{D}$ to all nodes in the reduced graph. The assignment to constant nodes is fixed by the rule; the search is for assignments to the remaining variable nodes. An assignment is *consistent* if it satisfies the property that if values $\ell, \ell' \in \mathbb{D}$ are respectively assigned to nodes n and n', which are connected by a p-edge, then the tuple $p(\ell, \ell')$ must be in $\mathcal{D}$.

As a less theoretical approach for deducing q-facts, we represent the domain $\mathbb{D}$ as a directed edge-colored graph $G_{\mathbb{D}} = (V_{\mathbb{D}}, E_{\mathbb{D}})$, where $V_{\mathbb{D}} = \mathbb{D}$ and $E_{\mathbb{D}}$ is constructed as follows. There is a p-colored edge $(\ell, \ell') \in E_{\mathbb{D}}$ for each fact $p(\ell, \ell') \in \mathcal{D}$. The directionality of the edges are set in the same manner as in expansion graphs, i.e., according to predicate constraints.

Now, given a computational path g we can find consistent assignments to nodes of g by simply traversing the graph $G_{\mathbb{D}}$. Precisely, consider a p-colored edge (n, n') of g. Let $\textbf{consts}_n \subset \mathbb{D}$ be the set of values assigned to the node n. Then, the set of values $\textbf{consts}_{n'}$ which will be assigned to the node n' is found by examining all the p-edges of graph $G_{\mathbb{D}}$ which start at nodes $\textbf{consts}_n$. In particular, a value $c' \in \mathbb{D}$ will be assigned to the node n' if there is a p-colored edge $(c, c') \in E_{\mathbb{D}}$ such that $c \in \textbf{consts}_n$. The above description gives rise to a process by which a consistent assignment to g nodes is found. This process is nothing else than BFS traversal of $G_{\mathbb{D}}$ which starts at the source of the computational path g and terminates after k steps, where k is the length of g.

Consider all nodes in all possible reduced expansion graphs of a DATALOG program Π. Let $\mathbf{consts}^*(\Pi)$ be the set of all different values from $\mathbb{D}$ which can be consistently assigned to these nodes. To complete this section we would like to show that $\mathbf{consts}^*(\Pi)$ is finite whenever Π is weakly safe and a database $\mathcal{D}$ is founded. To do so we prove that the set of values reachable from the sources of all the reduced expansion graphs in $G_{\mathbb{D}}$ is finite. Let $\mathbf{sources}^*(\mathcal{D})$ denote this set. Then,

Lemma 7.5. *If $\mathcal{D}$ is founded, then* $\mathbf{sources}^*(\mathcal{D})$ *is finite.*

Proof. The resulting graph $G_{\mathbb{D}}$ has a bounded out-degree, because the directionality of the edges is according to finiteness constraints and because there are finitely many predicates. Also, the number of sources in all the expansion graphs is finite, therefore it is finite in all the reduced such graphs (see claim (i) in Section 7.3). Now consider a search of BFS algorithm on graph $G_{\mathbb{D}}$ to reveal the reachable values $\mathbf{sources}^*(\mathcal{D})$ from a finite set of sources. After each step during the run, the number of exposed values is finite. Such search with bounded out-degree can reach an unbounded number of values, only if it can progress infinitely. This is prevented by the fact that $\mathcal{D}$ is founded. □

Finally, we show that for any program Π

$$\mathbf{consts}^*(\Pi) \subseteq \mathbf{sources}^*(\mathcal{D}). \tag{7.2}$$

Informally, consider a computational path g in some reduced expansion graph. Then, the BFS search on $G_{\mathbb{D}}$ must have revealed all the values of assignment to the nodes of g since it traversed *all* the outgoing edges of nodes unconditionally of their color.

It follows from (7.2) and Lemma 7.5 that the number of different assignments to target nodes in reduced expansion graphs is finite. Therefore, there is a finite number of such graphs which generate all these assignments, thereby completing the proof of Theorem 7.1.

Chapter 8

Computability

Having shown that every weakly safe program defined over founded database is safe, it is only natural to ask how such programs may be evaluated. After describing our computation model, this section discusses which programs may be computed in this model.

We continue to assume that all EDB predicates are binary. Let p be such predicate, $\mathcal{D}$ be a database. Then, similarly to [21], we assume the following:

- given constants (c, c'), one can determine in finite time whether $p(c, c') \in \mathcal{D}$,
- if $\{p_1\} \rightsquigarrow \{p_2\}$ (resp. $\{p_2\} \rightsquigarrow \{p_1\}$), then given a constant c, it is also possible to find in finite time all constants c' such that $p(c, c') \in \mathcal{D}$ (resp., $p(c', c) \in \mathcal{D}$), and
- if $\emptyset \rightsquigarrow \{p_1\}$ (resp. $\emptyset \rightsquigarrow \{p_2\}$), then we can find in finite time all constants c, such that there exists a constant c' for which $p(c, c') \in \mathcal{D}$ (resp. $p(c', c) \in \mathcal{D}$).

Note that in the absence of constraints, one cannot find in finite time all constants c' such that $p(c, c') \in \mathcal{D}$, (nor respectively, $p(c', c) \in \mathcal{D}$).

We say that Π is *computable* if there is an algorithm, that only accesses the database according to the rules of our computational model and correctly computes the result of Π in finite time. It may first seem that if a program is safe, then it is also computable. In this section we prove that safety is a necessary but not a sufficient requirement for computability of a DATALOG program and present a theorem which settles the computability problem for safe programs defined over founded databases. Then, Chapter 9 presents an evaluation algorithm for safe programs.

To see that safe programs are not necessarily computable, consider the following program Π, which is aimed to find all the parents $\mathtt{y}$ on which parent 'Bill' depends.

$$\begin{aligned}
&\tau_1\colon\ \mathsf{parents}(\mathtt{x},\mathtt{y}) \leftarrow \mathsf{child}(\mathtt{w},\mathtt{x}), \mathsf{child}(\mathtt{z},\mathtt{y}).\\
&\tau_2\colon\ \mathsf{dep_parents}(\mathtt{x},\mathtt{y}) \leftarrow \mathsf{dependant}(\mathtt{x},\mathtt{y}), \mathsf{parents}(\mathtt{x},\mathtt{y}).\\
&\tau_3\colon\ \mathsf{q}(\mathtt{y}) \leftarrow \mathsf{dep_parents}(\text{'Bill'},\mathtt{y}).
\end{aligned} \tag{8.1}$$

Then,

$$\mathcal{C}_\Pi = \{\{\mathsf{dep\_parents}_1\} \rightsquigarrow \{\mathsf{dep\_parents}_2\}, \emptyset \rightsquigarrow \{\mathsf{q}_1\}\}$$

can be inferred with the supposition that child and dependant predicates satisfy the following constraints:

$$\begin{aligned}
\mathcal{C} = \big\{ &\{\mathsf{child}_1\} \rightsquigarrow \{\mathsf{child}_2\},\\
&\{\mathsf{dependant}_1\} \rightsquigarrow \{\mathsf{dependant}_2\} \big\}.
\end{aligned}$$

Since $\mathcal{C}_\Pi$ contains $\emptyset \rightsquigarrow \{\mathsf{q}_1\}$, we have (Theorem 6.1) that this program is weakly safe. Also, since $\mathcal{D}$ is founded, it is also terminating by Theorem 7.1. Nevertheless, it is impossible to compute its output, because for any assignment μ to $\mathbb{D}$ and an atom $t = \mathsf{parents}(\mathtt{x},\mathtt{y})$ it is undecidable whether $\mu(t)$ holds. The difficulty here is that the evaluation process must find a child of $\mathtt{x}$ and a child of $\mathtt{y}$ to prove that $\mathtt{x}$ and $\mathtt{y}$ are indeed parents. Now, if the evaluation algorithm does find such children, it can conclude that $\mathsf{parents}(\mathtt{x},\mathtt{y})$ holds. But, what should the algorithm do if it does *not* find any children of $\mathtt{x}$?

Missing such evidence may be due to the fact that $\mathtt{x}$ indeed does not have children. Still, lack of evidence, could be a result of evaluation procedure's failure to explore the infinite database. In the genealogical interpretation, it could be that $\mathtt{x}$ has not yet given birth to children, or that these children exist but in a remote part of the universe. In the software interpretation, it could be that a software engineer, in a very remote galaxy, has implemented a class that inherits from $\mathtt{x}$, but the evaluation algorithm did not have sufficient resources to find this inheriting class.

If one is willing to permit similar existential queries in the algorithm, then any program which is terminating can also be evaluated. Under our computational model existential queries are not always allowed. Hence, it is sometimes not possible to evaluate even a terminating program. To preclude such queries from DATALOG programs we restrict program's predicates to be "recognizable", i.e., given $c_1, \ldots, c_k$ constants for a k-ary

predicate p, it is possible to decide whether $p(c_1, \ldots, c_k)$ is a fact. Note that every EDB predicate (trivially) satisfies this property. The following definition states it formally for IDB predicates.

Definition 8.1. *Predicate p is* variable-bound *if for every rule $r \in \Pi$ defining p with head h it holds that $\mathcal{C}_\Pi \models_r \mathbf{vars}(h) \rightsquigarrow \mathbf{terms}(r)$.*

For example, predicate dep_parents in example (8.1) is variable-bound since τ_2 contains only head's variables in the body, i.e., $\mathcal{C}_\Pi \models_{\tau_2} \mathbf{vars}(\mathrm{head}(\tau_2)) \rightsquigarrow \mathbf{terms}(\tau_2)$ trivially holds. It is easy to see that the goal predicate q is variable-bound as well. On the other hand, the predicate parents is not variable-bound. Observing its rule we can see that the head's variables do not imply any additional variables of the body. And, indeed, as explained earlier, during program computation one will not be able to decide whether $\mathsf{parents}(c_1, c_2)$ is a fact for any constants c_1, c_2. This is precisely the reason why the program of example (8.1) is not computable.

Therefore, we conclude that for a program to be computable *all* its predicates must be variable-bound as stated in the following theorem.

Theorem 8.1. *A program Π is* computable *if* (i) *it is safe and* (ii) *every predicate p appearing in it is variable-bound.*

The next chapter will present the evaluation algorithm for safe programs in which all the predicates are variable-bound thus proving Theorem 8.1.

Chapter 9

A Top-Down Evaluation Algorithm

This section describes a top-down algorithm for query evaluation. The heart of our algorithm is in function idb_eval (Algorithm 4), whose parameters include a predicate p, and a *subquery* expressed as a relation (in the relational algebra sense) Q, defining a possibly partial assignment to the positions of p, i.e., $\mathbf{scheme}(Q) \subseteq \mathbf{pos}(p)$.[1] The function answers the subquery by returning a relation whose columns are those columns in $\mathbf{pos}(p)$ which are finitely constrained by $\mathbf{scheme}(Q)$ and whose tuples are computed from the tuples of Q by these finiteness constraints.

For each of the rules defining the predicate, function idb_eval calls function rule_eval (Algorithm 5), which in its turn, calls function atom_eval (Algorithm 6) for each of the atoms in the rule. If the atom's predicate is an EDB, then atom_eval invokes edb_eval; otherwise, it recursively calls idb_eval.

Even simple rules such as $\mathsf{anc}(\mathtt{x},\mathtt{y}) \leftarrow \mathsf{anc}(\mathtt{x},\mathtt{z}), \mathsf{anc}(\mathtt{z},\mathtt{y})$, typical to transitive closure computation, may cause a naive implementation of idb_eval to recurse indefinitely. To guard against this predicament, the algorithm passes through the recursive calls variable $\mathbf{X}$, which stores in it all "open queries" in the recursion stack. Variable $\mathbf{X}$ is implemented as an associative array of relations. For each positions set $\mathbf{q}$, $\mathbf{q} \subseteq \mathbf{pos}(p)$, p an IDB predicate, $\mathbf{X}[\mathbf{q}]$ is a relation with scheme $\mathbf{q}$ containing all subqueries whose pattern is $\mathbf{q}$ which are on the recursion stack. At its 2^{nd} line, idb_eval restricts its interest to new such queries. At line 3, the function records the currently executing queries in $\mathbf{X}$.

Thus, the call to idb_eval that starts the evaluation process is with parameters:

[1] Recall that in our notation a "position" is not just an ordinal; it records also a predicate name. This is the reason that a set of positions can also be thought of as a scheme of a relation.

Algorithm 4 idb_eval(p, Q, $\mathbf{X}$)

1: **Let q** $\leftarrow$ **scheme**(Q) // *elicit the pattern of this query*
2: **Let** $Q' \leftarrow Q \setminus \mathbf{X}[\mathbf{q}]$ // *restrict interest to new queries*
3: $\mathbf{X}[\mathbf{q}] \leftarrow \mathbf{X}[\mathbf{q}] \cup Q'$ // *record remaining queries in cache*
4: **Let m** be the maximal set s.t. $\mathbf{q} \rightsquigarrow \mathbf{m} \in \mathcal{C}_\Pi$ // **m** *is the scheme of the answer relation*
5: **If** $Q' \neq \emptyset$ **then** // *queries remained for execution*
6: **Repeat** // *exercise all rules until no new answers are found*
7: **For all** $r \in \Pi$ such that $\mathrm{pred}(r) = p$ **do** // *try rule* r
8: **Let** $T \leftarrow$ rule_eval(r, pos2term$_r(Q')$, $\mathbf{X}$)
9: $\mathbf{M}[\mathbf{q}] \leftarrow \mathbf{M}[\mathbf{q}] \cup \pi_{\mathbf{m}}$term2pos$_r(T)$
10: **end for**
11: **until** no changes in $\mathbf{M}$
12: **end If**
13: **return** $\mathbf{M}[\mathbf{q}] \bowtie Q$ // *restrict global answer set to queries in* Q

1. q — the program goal,
2. I — the relation with no columns and a single, empty, tuple and
3. $\mathbf{X}$ in which all entries are initialized to an empty relation.

In addition to $\mathbf{X}$, the algorithm maintains a similarly organized *global* array $\mathbf{M}$ for results *memoization*, except that the scheme of $\mathbf{M}[\mathbf{q}]$ is **m**, where **m** is the maximal set such that $\mathbf{q} \rightsquigarrow \mathbf{m}$. The main loop of idb_eval (lines 6–11) uses the results of calls to rule_eval to extend, as long as this is possible, relation $\mathbf{M}[\mathbf{q}]$. The function result is obtained by restricting $\mathbf{M}[\mathbf{q}]$ (which records *all* queries of pattern **q** that the algorithm *ever* executed) to answers of queries in Q; this is carried out by the natural join operation in line 13.

In order to delegate its work to function rule_eval, function idb_eval must translate the query Q', which is formulated in terms of positions in p, to the list of symbolic variables that rule r expects. To this end, we use an overloaded version of function pos2term$_r$ (invoked at line 8), which returns its input relation with renamed columns as per the head of rule r. The reverse translation of rule_eval's return value, is carried out by the call to (an overloaded version of) function term2pos$_r$, at line 9. This line also projects the return value into the scheme **m**.

Consider now function rule_eval, depicted in Algorithm 5. Line 1 of this function begins the computation of a subquery with respect to a rule, by augmenting the given

Algorithm 5 rule_eval(r, Q, $\mathbf{X}$)

1: $Q \leftarrow Q \bowtie \mathit{CONSTS}_r$
2: **Repeat**
3: **For all** $a \in$ **body**(r) **do**
4: $Q \leftarrow Q \bowtie$ atom_eval(a, $\pi_a Q$, $\mathbf{X}$)
5: **end for**
6: **until** no changes in Q
7: **return** $\pi_{\mathrm{head}(r)} Q$

subquery with the values of constants used in the rule: variable CONSTS_r denotes the relation whose column names are simply **consts**(r), while its single tuple consists also of these constants. The recursive call to atom_eval (line 4) is preceded by a projection to the variables (and constants) used in the atom. We assume that the operator π ignores columns in projection scheme which do not exist in the projected relation. Hence, the projection succeeds even if Q does not contain all terms of the current atom. (In particular, if Q does not contain any term of a, a no-columns relation containing the empty tuple is returned.) A projection to terms in the rule head is applied before the function is returned.

Finally take note of Algorithm 6, depicting function atom_eval. This function is rather straightforward; note however that the recursive call to functions that evaluate a predicate requires a change of vocabulary, prior to, and after the call.

Algorithm 6 atom_eval(a, Q, $\mathbf{X}$)

1: **Let** $Q' \leftarrow \mathsf{term2pos}_a(Q)$ // *bound positions of* a
2: **Let** $p \leftarrow \mathrm{pred}(a)$ // *elicit the predicate of this atom*
3: **Let** $A \leftarrow \begin{cases} \mathsf{edb\_eval}(p, Q') & \text{if } p \in \mathbf{edb}(\Pi) \\ \mathsf{idb\_eval}(p, Q', \mathbf{X}) & \text{otherwise} \end{cases}$
4: **return** $\mathsf{pos2term}_a(A)$

9.1 Some Intuition

Before marching on to the proving the correctness of the evaluation algorithm, we are inclined to say few words on the repetitive process by which atoms are examined or re-examined. The gained intuition should make the constructions used in the proof clearer.

Our algorithm differs from the original QSQR algorithm [26, 27] in the way that the evaluation of a rule is carried out. In particular, function rule_eval iterates over the rule's atoms (lines 2–6) until all the variables of a rule are bound. (Note that in general, the evaluation of a program may complete without binding all variables of a rule, even if a rule is used in the evaluation process.) Unlike the original algorithm in which only one pass is enough to bind all the rule's variables, the iterative process of Algorithm 5 is such that atoms of a specific rule may be evaluated for many times in the course of the same invocation of rule_eval. This difference is due to the restrictions placed on the computational model by the infiniteness of the database. As explained in Chapter 8, we cannot arbitrarily retrieve data even from EDB predicates. Such retrieval is restricted to the known finiteness constraints. Thus, in many cases during the program evaluation we will be able to evaluate an atom a only partially, i.e., only values for a subset of a's positions will be retrieved. Later we can re-evaluate a, if we have additional bindings on more of its positions. The following example demonstrates.

Example 9.1.

$$\begin{aligned} &\mathsf{q}(\mathtt{y},\mathtt{z}) \leftarrow \mathsf{a}(\mathtt{y},\mathtt{z}), \mathsf{p}(\text{`}c\text{'},\mathtt{y},\mathtt{z}). \\ &\mathsf{p}(\mathtt{x},\mathtt{y},\mathtt{z}) \leftarrow \mathsf{a}(\mathtt{x},\mathtt{y}), \mathsf{b}(\mathtt{y},\mathtt{z}). \end{aligned} \tag{9.1}$$

Let a *and* b *be EDB predicates. Suppose also that we have only one constraint on the database:* $\{\mathsf{a}_1\} \rightsquigarrow \{\mathsf{a}_2\}$ *. It is not difficult to prove that this program is weakly safe. If the database is founded, then this program is also safe.*

Now, suppose that we would like to evaluate q *with our algorithm. We start with* idb_eval *which in turn delegates the work to function* rule_eval *which receives in its first argument the only rule defining* q*. The main loop then iterates over the atoms of* q*-rule for three times as described next:*

1. The 1^{st} evaluation of atom $\mathsf{a}(\mathtt{y},\mathtt{z})$ *produces the empty tuple only, since we cannot retrieve data from the EDB predicate* a *without binding the first position;*

2. The 1^{st} evaluation of atom $\mathsf{p}(\text{`c'},\mathtt{y},\mathtt{z})$ *triggers* idb_eval *function which in turn leads to* rule_eval *function. Then, given the binding of* x *to 'c', we can compute all values* y *that satisfy the first atom* $\mathsf{a}(\mathtt{x},\mathtt{y})$*. However, we cannot find the corresponding* z *values, since there are no constraints at all in* b*. This evaluation is given by*

the following relational algebra expression

$$\sigma_{\$1=\text{`}c\text{'}}\mathsf{a}.$$

3. The 2^{nd} evaluation of atom $\mathsf{a}(\mathtt{y},\mathtt{z})$ *produces bindings for* $\mathtt{z}$ *since now there exist bindings to* $\mathtt{y}$. *Now, the assignments known for* $\mathtt{y}$ *and* $\mathtt{z}$ *can be summarized by*

$$\pi_{\$3,\$4}\big(\sigma_{\$2=\$3\wedge\$1=\text{`}c\text{'}}\big((\mathsf{a}\times\mathsf{a})\big)\big).$$

4. The 2^{nd} evaluation of atom $\mathsf{p}(\text{`c'},\mathtt{y},\mathtt{z})$ *retains only the bindings which satisfy the atom* $\mathsf{b}(\mathtt{y},\mathtt{z})$. *The following expresses the final assignments:*

$$\pi_{\$3,\$4}\big(\sigma_{\$3=\$6\wedge\$4=\$8}\big(\sigma_{\$2=\$3\wedge\$1=\text{`}c\text{'}}(\mathsf{a}\times\mathsf{a})\big)\times\big(\sigma_{\$2=\$3\wedge\$1=\text{`}c\text{'}}(\mathsf{a}\times\mathsf{b})\big)\big).$$

5. The 3^{rd} iteration over the rule body *verifies that the previously found bindings satisfy both atoms. At this stage all bindings remain.*

The above process can be naturally thought of as an evaluation of a slightly different program from the QSQR point of view, i.e., it looks like the original algorithm was executed on a different input.

In particular, step 2 in the above, namely the preliminary partial evaluation of atom

$$\mathsf{p}(\text{`c'},\mathtt{y},\mathtt{z}),$$

can be thought of as the evaluation of another atom

$$\mathsf{p'}(\text{`c'},\mathtt{y})$$

where $\mathsf{p'}$ is a predicate we introduce together with the rule

$$\mathsf{p'}(\mathtt{x},\mathtt{y}) \leftarrow \mathsf{a}(\mathtt{x},\mathtt{y}).$$

All in all, the slightly different program which describes the above process is as follows:

$$\mathsf{q}(\mathtt{y},\mathtt{z}) \leftarrow \mathsf{p'}(\text{`c'},\mathtt{y}),\mathsf{a}(\mathtt{y},\mathtt{z}),\mathsf{p}(\text{`c'},\mathtt{y},\mathtt{z}).$$
$$\mathsf{p'}(\mathtt{x},\mathtt{y}) \leftarrow \mathsf{a}(\mathtt{x},\mathtt{y}).$$
$$\mathsf{p}(\mathtt{x},\mathtt{y},\mathtt{z}) \leftarrow \mathsf{a}(\mathtt{x},\mathtt{y}),\mathsf{b}(\mathtt{y},\mathtt{z}).$$

The correctness proof, provided in the reminder of this chapter, uses such programs called *adapted programs* to represent reformulation of the evaluation process. Precisely, for a DATALOG program Π the proof introduces an adapted program $\widetilde{\Pi}$ for which it holds that the run of our evaluation algorithm (Algorithm 4) on Π is the same as the run of QSQR algorithm on $\widetilde{\Pi}$. Although we have not presented here the original QSQR algorithm, the *main* difference between it and our evaluation algorithm is in Algorithm 5, i.e., evaluation of a rule.

Adapted programs must have the following properties:

1. rule evaluation requires only one iteration to bind all the rule's variables (QSQR algorithm goes over rule's atoms only once), and

2. atom evaluation must be able to find full bindings from the known ones (QSQR algorithm is not aware of the restrictive computational model).

Before we formally present a construction of adapted programs, let us discuss in further detail what happens in function rule_eval. Line 4 in the above function evaluates every atom of the input rule unconditionally. Some of the evaluations may be unnecessary in a sense that they do not change the value of variable Q. We state that we can determine statically which atom evaluations are required and which are superfluous. To do so we present the following lemma which identifies the way in which Q changes during the run of rule_eval function.

Lemma 9.1. *Let* $a \in$ **body**(r) *be an examined atom in the body of* rule_eval *function (line 4). Let* $\boldsymbol{X} =$ **scheme**$(Q) \cap$ **terms**(a) *be the set of terms which are common to atom* a *and* Q*'s scheme. Let* $\boldsymbol{Y}$ *be the set of* a*'s terms such that* $\mathsf{term2pos}_a(\boldsymbol{X}) \rightsquigarrow \mathsf{term2pos}_a(\boldsymbol{Y}) \in \mathsf{min}(\mathcal{C}_\Pi)$. *Then, after evaluating the atom (line 4), it holds that the new value of* **scheme**(Q) *is* **scheme**$(Q) \cup \boldsymbol{Y}$.

The proof can be done by induction on the recursive execution of algorithm functions.

Now it is possible to recognize cases in which atom evaluations are useless and thus will not be represented in adapted programs.

Precisely, let $a \in$ **body**(r) be an examined atom (see line 4). Let $\mathbf{X} =$ **scheme**$(Q) \cap$ **terms**(a) be the set of terms which are common to a and Q's scheme. Then, the following conditions identify the cases in which evaluation of a is unnecessary.

1. $\mathbf{X} = \emptyset$. Here the evaluation of a is irrelevant to the currently bound variables. An example of such a case is step 1 (the evaluation of $\mathsf{a}(\mathtt{y}, \mathtt{z})$ for the first time).

2. $\mathbf{X} \neq \mathbf{terms}(a)$ and there does not exist $\mathbf{Y} \subseteq \mathbf{terms}(a)$ such that $\mathsf{term2pos}_a(\mathbf{X}) \rightsquigarrow \mathsf{term2pos}_a(\mathbf{Y})$ and $\mathbf{X} \subset \mathbf{Y}$ ($\mathbf{X} \neq \mathbf{Y}$). In this case the set of relevant bound variables $\mathbf{X}$ do not bind any variable of $\mathbf{terms}(a)$.

To summarize this discussion we present observations which lie in the basis of adapted program construction.

Observation 9.2. *An atom can be evaluated many times during the evaluation of the rule containing it — each time with a new set of bound variables.*

Our algorithm iteratively evaluates body atoms until it reaches a fixpoint (see the rule_eval function), i.e., the algorithm tries to expand the set of bound variables as much as possible. Each time the same atom is examined, the set of bound variables may grow. Hence, some atoms may be evaluated more than once with a new set of bound variables thus initiating repeated rule evaluations. Therefore, the adapted program must reflect these rule evaluations by constructing a dedicated adapted rule for each such case.

For example, in the adapted program of Example 9.1, one auxiliary predicate p' was defined and the q-rule was adapted to represent atom evaluations to be performed in order to bind all the rule's variables in a single iteration. Note that the atoms in the adapted q rule were rearranged to achieve the "one-iteration" evaluation.

Observation 9.3. *The application of function* rule_eval *to a rule r, where terms $\boldsymbol{X}$ in the rule head are known, terminates when terms $\boldsymbol{Y} \subseteq \mathbf{vars}(r)$ are bound, where $\boldsymbol{Y}$ is the maximal set such that $\mathcal{C}_\Pi \models_r \boldsymbol{X} \rightsquigarrow \boldsymbol{Y}$.*

Consider an atom a to be evaluated and a set of bounded positions $\mathbf{x} \subseteq \mathbf{pos}(p)$, where $p = \mathrm{pred}(a)$. Then, according to the algorithm, it triggers rule evaluation for all the rules r defining p with bound variables $\mathbf{X} = \mathsf{pos2term}_r(\mathbf{x})$. According to Lemma 9.1, the function rule_eval expands $\mathbf{scheme}(Q)$ in a similar way the function closure does (Algorithm 1). Thus, before rule evaluation takes place, we know exactly which rule variables will become bound afterwards. They are precisely the maximal set $\mathbf{Y}$, such that $\mathcal{C}_\Pi \models_r \mathbf{X} \rightsquigarrow \mathbf{Y}$.

9.2 Correctness Proof

The reminder of this chapter is devoted to prove that our evaluation algorithm is correct. For this purpose, we shift the attention to the original QSQR evaluation algorithm which is known to be correct. As explained before, we accomplish this by introducing the adapted program $\widetilde{\Pi}$ which is executed by QSQR algorithm thus representing the run of our evaluation algorithm on the original program Π. By showing that such adapted program is equivalent to its original program, i.e. produces the same output as Π, we prove that our evaluation algorithm is correct. The previous sentence would have been true if the database was finite. In our settings, QSQR algorithm may not terminate as well. Thus, showing the above equivalence only proves that for each fact f returned by our evaluation algorithm it holds that $f \in \Pi_q^\infty(\mathcal{D})$. By proving that the run terminates, we show the opposite direction, i.e., that every fact $f \in \Pi_q^\infty(\mathcal{D})$ is also returned by our evaluation algorithm.

9.2.1 Construction of the Adapted Program

Next we define the construction of auxiliary predicates which are necessary for the adapted program. Let p be an IDB predicate. Let $\mathbf{x} \subseteq \mathbf{pos}(p)$ be a set of p-positions and $\mathbf{y} \subseteq \mathbf{pos}(p)$, $\mathbf{x} \subseteq \mathbf{y}$ be the maximal set such that $\mathbf{x} \rightsquigarrow \mathbf{y} \in \mathcal{C}_\Pi$ holds. Then, an *auxiliary predicate* of p with respect to $\mathbf{x}$, denoted $p^{\mathbf{x},\mathbf{y}}$, is a predicate of arity $|\mathbf{y}|$. (A predicate name $p^{\mathbf{x},\mathbf{y}}$ implies that this predicate will be used to evaluate p-predicate when $\mathbf{x}$ are the only bounded positions.) The rules of $p^{\mathbf{x},\mathbf{y}}$ are the "adapted" rules of p, constructed as described next.

Consider a p-rule $r \in \Pi$. Let $\mathbf{X} = \mathsf{pos2term}_r(\mathbf{x})$ and $\mathbf{Y} = \mathsf{pos2term}_r(\mathbf{y})$, i.e., $\mathbf{X}$ is the terms rewrite of $\mathbf{x}$ and $\mathbf{Y}$ is the terms rewrite of $\mathbf{y}$. Then, the adapted rule of r, denoted $r^{\mathbf{x},\mathbf{y}}$ has the head atom

$$p^{\mathbf{x},\mathbf{y}}(\mathbf{Y}_1, \ldots, \mathbf{Y}_k)$$

while the body is constructed by simulating the run of a slightly different version of function closure (Algorithm 1) on r, $\mathbf{X}$ and the minimized constraints set $\mathsf{min}(\mathcal{C}_\Pi)$. The difference is in the procedure for choosing a constraint to expand $\mathbf{X}^+_{\mathsf{min}(\mathcal{C}_\Pi),r}$ (see line 5). For further references, we will refer to Algorithm 1 with a new choosing procedure as the changed Algorithm 1. To prevent confusion, let $\mathbf{Y}'$ denote the value for $\mathbf{Y}$ variable

used by the algorithm. The new choosing procedure is different only in its 2^{nd} condition. Precisely, we choose $\mathbf{Y}'$ such that

$$\mathbf{Y}' = \mathbf{X}^{+}_{\mathsf{min}(\mathcal{C}_\Pi),r} \cap \mathbf{terms}(a_i).$$

Then, when working with the minimized version $\mathsf{min}(\mathcal{C}_\Pi)$ of $\mathcal{C}_\Pi$, the value for $\mathbf{Z}$ is unique.

The adapted rule $r^{\mathbf{x},\mathbf{y}}$ contains only the variables implied by $\mathbf{X}$. The i^{th} atom that expanded the set $\mathbf{X}^{+}_{\mathsf{min}(\mathcal{C}_\Pi),r}$ will be the i^{th} atom in the adapted rule body. Moreover, if not all the atom's variables could be constrained, we replace it with an atom of a new auxiliary predicate. To define precisely the body of the adapted rule, we introduce the notion of a *closure sequence*, which represents the run of Algorithm 1 on a given rule.

Definition 9.4. *Let r be a rule, $\mathbf{x} \subseteq \mathbf{pos}(\mathrm{pred}(r))$ and $\mathcal{C}' = \mathsf{min}(\mathcal{C}_\Pi)$ a set of constraints. A* closure sequence *of r with respect to $\mathbf{x}$ and $\mathcal{C}'$, denoted $\mathbf{cls}(r, \mathbf{x}, \mathcal{C}')$ is a sequence of pairs (a, σ) where atom $a \in \mathbf{body}(r)$ and constraint $\sigma \in \mathcal{C}'$.*

A pair $(a, \sigma)_i$ is the i^{th} pair in the sequence if at the i^{th} execution of line 6 of the changed function closure*$(r, \mathcal{C}', \boldsymbol{X})$, where $\boldsymbol{X} = \mathsf{pos2term}_a(\mathbf{x})$, σ is the constraint over a-predicate which expanded $\boldsymbol{X}^{+}_{\mathcal{C}',r}$, i.e.,*

$$\sigma = \mathsf{term2pos}_a(\boldsymbol{Y}) \rightsquigarrow \mathsf{term2pos}_a(\boldsymbol{Z}).$$

For example, let r be the following q-rule

$$\mathsf{q}(\mathtt{x}, \mathtt{y}, \mathtt{z}) \leftarrow \mathsf{p}(\mathtt{x}, \mathtt{u}, \mathtt{w}, \mathtt{v}), \mathsf{e}(\mathtt{u}, \mathtt{w}), \mathsf{s}(\mathtt{v}, \mathtt{z}, \mathtt{y}). \tag{9.2}$$

in which p and s are IDB predicates and e is an EDB predicate. Assume that the following constraints are part of $\mathsf{min}(\mathcal{C}_\Pi)$

$$\{\mathsf{p}_1\} \rightsquigarrow \{\mathsf{p}_1, \mathsf{p}_2\}, \{\mathsf{p}_3\} \rightsquigarrow \{\mathsf{p}_3, \mathsf{p}_4\}, \{\mathsf{e}_1\} \rightsquigarrow \{\mathsf{e}_2\}, \{\mathsf{s}_1\} \rightsquigarrow \{\mathsf{s}_1, \mathsf{s}_3\}.$$

Then, the closure sequence of r with respect to $\mathsf{min}(\mathcal{C}_\Pi)$ and $\{1\}$ is

$$\begin{aligned}\mathbf{cls}(r, \{1\}, \mathsf{min}(\mathcal{C}_\Pi)) = \langle &(\mathsf{p}(\mathtt{x}, \mathtt{u}, \mathtt{w}, \mathtt{v}), \{1\} \rightsquigarrow \{1, 2\}), (\mathsf{e}(\mathtt{u}, \mathtt{w}), \{1\} \rightsquigarrow \{2\}), \\ &(\mathsf{p}(\mathtt{x}, \mathtt{u}, \mathtt{w}, \mathtt{v}), \{3\} \rightsquigarrow \{3, 4\}), (\mathsf{s}(\mathtt{v}, \mathtt{z}, \mathtt{y}), \{1\} \rightsquigarrow \{1, 3\})\rangle.\end{aligned} \tag{9.3}$$

Given a closure sequence, we can now construct an adapted body rule. In particular, each element of the closure sequence defines a new body atom of the adapted rule, which

may refer to an auxiliary predicate. Specifically, consider an element $(a, \mathbf{x} \rightsquigarrow \mathbf{y})$ of the closure sequence in which $\mathrm{pred}(a)$ is an IDB predicate. Then, we create an auxiliary predicate $\mathrm{pred}(a)^{\mathbf{x},\mathbf{y}}$ which is defined by the adapted rules of $\mathrm{pred}(a)$. Each such rule r retains in its head only the variables $\mathsf{pos2term}_r(\mathbf{y})$ and its body contains only variables $\mathbf{Z}$ that are implied by the variables in the head. (Note that to create such rules, one may have to recursively define auxiliary predicates.)

Usually, when a closure sequence element refers to an EDB predicate, no auxiliary predicates need to be created. However, there is one case in which a new *EDB* predicate must be introduced. This happens when a closure sequence element is of the form $(a, \emptyset \rightsquigarrow \mathbf{y})$ and it holds that $|\mathbf{y}| = 1$. Let $p = \mathrm{pred}(a)$ and $\mathbf{y} = \{p_i\}$. Then, we create a new unary EDB predicate denoted p^i. The predicate p^i contains a fact $p^i(c)$ if there exists a constant c' such that $p(c, c')$ (resp. $p(c', c)$) holds when $i = 1$ (resp. $i = 2$). Note that the resulting predicate contains finitely many facts, which can be computed according to our computation model.

Now we define the notion of an adapted rule formally.

Definition 9.5. *Let $r \in \Pi$ be a p-rule and $\mathbf{x} \subseteq \mathbf{pos}(r)$ and let $\mathbf{x} \rightsquigarrow \mathbf{y} \in \mathsf{min}(\mathcal{C}_\Pi)$. Let $\boldsymbol{Y} = \mathsf{pos2term}_a(\mathbf{y})$ and $|\boldsymbol{Y}| = k$.*

Then, the adapted rule *$r^{\mathbf{x},\mathbf{y}}$ of r with respect to $\mathbf{x}$ is a rule with head*

$$p^{\mathbf{x},\mathbf{y}}(\boldsymbol{Y}_1, \ldots, \boldsymbol{Y}_k)$$

and body constructed as follows: for each element $(a, \mathbf{y} \rightsquigarrow \mathbf{z})_i$ in the closure sequence $\mathbf{cls}(r, \mathbf{x}, \mathsf{min}(\mathcal{C}_\Pi))$

1. *if a refers to an EDB predicate and either $|\mathbf{y}| > 1$ or $|\mathbf{z}| \geq 1$ holds, then the i^{th} atom is a itself,*
2. *otherwise, if atom a refers to an EDB predicate $s = \mathrm{pred}(a)$ and it holds that $\mathbf{y} = \emptyset$ and $|\mathbf{z}| = 1$ where $\mathbf{z} = \{s_j\}$, then the i^{th} atom of the adapted body is atom $s^j(\mathsf{z})$ such that $\mathsf{z} = \mathsf{pos2term}_a(\mathbf{z})$.*
3. *otherwise, atom a must refer to an IDB predicate $s = \mathrm{pred}(a)$. Then the i^{th} atom of the adapted body is atom $s^{\mathbf{y},\mathbf{z}}(\boldsymbol{Z}_1, \ldots, \boldsymbol{Z}_{|\mathbf{Z}|})$ where $\boldsymbol{Z} = \mathsf{pos2term}_a(\mathbf{z})$.*

In addition, for each atom $a \in \mathbf{body}(r)$ such that $\mathbf{terms}(a) \subseteq \boldsymbol{X}^+_{\mathsf{min}(\mathcal{C}_\Pi),r}$, the adapted body also contains the atom a itself if a refers to an EDB predicate, or

the atom $s^{\mathbf{pos}(s),\mathbf{pos}(s)}(\boldsymbol{U}_1,\ldots,\boldsymbol{U}_{\mathbf{pos}(s)})$ *such that* $s = \mathrm{pred}(a)$ *is an IDB predicate and* $\mathbf{terms}(a) = \boldsymbol{U}$.

These atoms are placed at the latest positions in an arbitrary order.

For example, assuming that $\mathcal{C}_\Pi$ does not contain any new p- or s- constraints, the adapted rule $r^{\{1\},\{1,2\}}$ from (9.2) is as follows

$$\begin{aligned}\mathsf{q}^{\{1\},\{1,2\}}(\mathtt{x},\mathtt{y}) \leftarrow & \mathsf{p}^{\{1\},\{1,2\}}(\mathtt{x},\mathtt{u}),\\ & \mathsf{e}(\mathtt{u},\mathtt{w}),\\ & \mathsf{p}^{\{3\},\{3,4\}}(\mathtt{w},\mathtt{v}),\\ & \mathsf{s}^{\{1\},\{1,3\}}(\mathtt{v},\mathtt{y}),\\ & \mathsf{p}^{\{1,2,3,4\},\{1,2,3,4\}}(\mathtt{x},\mathtt{u},\mathtt{w},\mathtt{v}),\\ & \mathsf{e}(\mathtt{u},\mathtt{w}).\end{aligned}$$

Here the first four atoms were created due to the closure sequence (9.3), while the last two atoms were added since their original atoms $a \in \mathbf{body}(r)$ satisfied $\mathbf{terms}(a) \subseteq \{1\}^+_{\mathsf{min}(\mathcal{C}_\Pi),r}$ as described in Definition 9.5.

Having defined the meaning of adapted rules and auxiliary predicates, we are ready to describe the construction of an adapted program. Informally, the construction starts by computing the adapted rules of q-rules with respect to the empty set. During their creation new auxiliary predicates might be created which in turn will lead to construction of their adapted rules. This process is continued until all the auxiliary predicates are defined. (The process terminates since for each predicate of arity k there may be at most 2^k auxiliary predicates.) Formally,

Definition 9.6. *The* adapted program *of* Π*, denoted* $\widetilde{\Pi}$*, is a program with the goal predicate* $q^{\emptyset,\mathbf{pos}(q)}$ *as defined by corresponding adapted rules (Definition 9.5).*

Note that for every rule $r \in \Pi$ there may be several corresponding adapted rules in $\widetilde{\Pi}$. Consider a case in which an IDB predicate p satisfies several non-trivial constraints (a constraint $\mathbf{x} \rightsquigarrow \mathbf{y}$ is trivial if $\mathbf{y} \subseteq \mathbf{x}$). Then, for every p-rule r there may be as many adapted rules as there are non-trivial constraints of p. Let $\mathcal{A}(r) \subseteq \widetilde{\Pi}$ denote the set of adapted rules corresponding to a rule r.

We complete this section with a full example of DATALOG program and its resulting adapted program.

Example 9.7. *Consider the following program* Π *in which* a *and* b *are EDB predicates:*

$$\tau_1\colon \mathsf{q}(\mathtt{y}) \leftarrow \mathsf{p}(\text{`}1\text{'}, \mathtt{y}, \mathtt{z}), \mathsf{a}(\text{`}2\text{'}, \mathtt{z}).$$
$$\tau_2\colon \mathsf{p}(\mathtt{x}, \mathtt{y}, \mathtt{z}) \leftarrow \mathsf{p}(\mathtt{u}, \mathtt{y}, \mathtt{x}), \mathsf{a}(\mathtt{z}, \mathtt{u}).$$
$$\tau_3\colon \mathsf{p}(\mathtt{x}, \mathtt{y}, \mathtt{z}) \leftarrow \mathsf{a}(\mathtt{x}, \mathtt{y}), \mathsf{b}(\mathtt{y}, \mathtt{z}).$$

Assume that $\mathcal{C} = \{\{\mathsf{a}_1\} \rightsquigarrow \{\mathsf{a}_2\}, \{\mathsf{b}_2\} \rightsquigarrow \{\mathsf{b}_1\}\}$.
By running Algorithm 3 for computing constraints of IDB predicates, we receive that

$$\{\mathsf{p}_1\} \rightsquigarrow \{\mathsf{p}_2\}, \{\mathsf{p}_3\} \rightsquigarrow \{\mathsf{p}_2\}, \emptyset \rightsquigarrow \{\mathsf{q}_1\}$$

are part of $\mathcal{C}_\Pi$.

Then, the adapted program $\widetilde{\Pi}$ *is as follows:*

$$\tau_1^{\emptyset,\{1\}}\colon \mathsf{q}^{\emptyset,\{1\}}(\mathtt{y}) \leftarrow \mathsf{p}^{\{1\},\{1,2\}}(\text{`}1\text{'}, \mathtt{y}), \mathsf{a}(\text{`}2\text{'}, \mathtt{z}), \mathsf{p}^{\{1,2,3\},\{1,2,3\}}(\text{`}1\text{'}, \mathtt{y}, \mathtt{z}), \mathsf{a}(\text{`}2\text{'}, \mathtt{z}).$$
$$\tau_2^{\{1\},\{1,2\}}\colon \mathsf{p}^{\{1\},\{1,2\}}(\mathtt{x}, \mathtt{y}) \leftarrow \mathsf{p}^{\{3\},\{2,3\}}(\mathtt{y}, \mathtt{x}).$$
$$\tau_3^{\{1\},\{1,2\}}\colon \mathsf{p}^{\{1\},\{1,2\}}(\mathtt{x}, \mathtt{y}) \leftarrow \mathsf{a}(\mathtt{x}, \mathtt{y}).$$
$$\tau_2^{\{3\},\{2,3\}}\colon \mathsf{p}^{\{3\},\{2,3\}}(\mathtt{y}, \mathtt{z}) \leftarrow \mathsf{a}(\mathtt{z}, \mathtt{u}), \mathsf{p}^{\{1\},\{1,2\}}(\mathtt{u}, \mathtt{y}).$$
$$\tau_3^{\{3\},\{2,3\}}\colon \mathsf{p}^{\{3\},\{2,3\}}(\mathtt{y}, \mathtt{z}) \leftarrow \mathsf{b}(\mathtt{y}, \mathtt{z}).$$
$$\tau_2^{\{1,2,3\},\{1,2,3\}}\colon \mathsf{p}^{\{1,2,3\},\{1,2,3\}}(\mathtt{x}, \mathtt{y}, \mathtt{z}) \leftarrow \mathsf{p}^{\{3\},\{2,3\}}(\mathtt{y}, \mathtt{x}), \mathsf{a}(\mathtt{z}, \mathtt{u}), \mathsf{p}^{\{1,2,3\},\{1,2,3\}}(\mathtt{u}, \mathtt{y}, \mathtt{x}), \mathsf{a}(\mathtt{z}, \mathtt{u}).$$
$$\tau_3^{\{1,2,3\},\{1,2,3\}}\colon \mathsf{p}^{\{1,2,3\},\{1,2,3\}}(\mathtt{x}, \mathtt{y}, \mathtt{z}) \leftarrow \mathsf{a}(\mathtt{x}, \mathtt{y}), \mathsf{b}(\mathtt{y}, \mathtt{z}).$$

For the reminder of this chapter we will adopt the following convention in our notation:

1. For a predicate $p \in \Pi$, we will use the notation $\overline{p}$ for $p^{\mathbf{pos}(p),\mathbf{pos}(p)} \in \widetilde{\Pi}$, i.e., the auxiliary predicate in $\widetilde{\Pi}$ used for testing membership.

2. For a rule $r \in \Pi$, we will use the notation $\overline{r}$ for the rule $r^{\mathbf{pos}(p),\mathbf{pos}(p)} \in \widetilde{\Pi}$, i.e., the adapted r-rule for testing membership.

In Example 9.7 the rule $\tau_3^{\{1,2,3\},\{1,2,3\}}$ will be denoted $\overline{\tau_3}$ according to the above convention.

9.2.2 Equivalence proof

This section is devoted to the proof that the adapted program is equivalent to the original one. To this end we show that both programs deduce the same facts for their goal predicates.

In the following, we will frequently need to refer to *projected set* of facts defined as follows. Let p be a predicate and $\mathbf{x} \subseteq \mathbf{pos}(p)$ a set of positions. Then, a projected p-fact is obtained from a p-fact by omitting from it all values in positions not in $\mathbf{x}$. If $\mathcal{F}$ is a set of p-facts, then $\pi_{\mathbf{x}}[\mathcal{F}]$ is the set of p-facts thus obtained from F. By abuse of terminology we refer to projected p-facts as p-facts as well; no confusion shall arise.

The following lemma defines the relation between a rule of the original program and its adapted version. The lemma states that the application of an adapted rule $r^{\mathbf{x},\mathbf{y}}$ to a set of "projected" facts deduces at least the same set of facts obtained by applying the original rule r to the original set of facts and then projecting the result on the positions $\mathbf{y}$.

Lemma 9.2. *Let $\mathcal{F} = \mathcal{F}_{edb} \cup \mathcal{F}_{idb}$ be a set of facts on the predicates of program Π where $\mathcal{F}_{edb}$ are the facts on EDB predicates and $\mathcal{F}_{idb}$ are the facts on IDB predicates. Let $\mathcal{F}'_{idb}$ be the set of facts obtained from $\mathcal{F}_{idb}$ by the following process: for each p-fact $f \in \mathcal{F}_{idb}$, and for each $p^{\mathbf{x},\mathbf{y}}$ such that $p^{\mathbf{x},\mathbf{y}} \in \mathbf{idb}(\widetilde{\Pi})$, the set $\mathcal{F}'_{idb}$ contains every fact obtained from projecting f on the positions $\mathbf{y}$. Let $\mathcal{F}' \supseteq (\mathcal{F}_{edb} \cup \mathcal{F}'_{idb})$. Let $r \in \Pi$ be a p-rule and $r^{\mathbf{x},\mathbf{y}} \in \mathcal{A}(r)$ be its adapted rule. Then,*

$$\pi_{\mathbf{y}}[r_p(\mathcal{F})] \subseteq r^{\mathbf{x},\mathbf{y}}(\mathcal{F}')$$

for every set of facts $\mathcal{F}$.

Proof. Let $f \in r_p(\mathcal{F})$. According to the semantics of Datalog rules, there exists an assignment $\mu : \mathbf{vars}(r) \to \mathbb{D}$ due to which f was deduced, i.e., for all atoms $a \in \mathbf{body}(r)$, it holds that $\mu(a) \in \mathcal{F}$. According to the construction of adapted rules and $\mathcal{F}'$ it follows that μ is an assignment such that $\mu(a') \in \mathcal{F}'$ for all $a' \in \mathbf{body}(r^{\mathbf{x},\mathbf{y}})$. The lemma follows. □

The next lemma shows that the adapted program deduces at least the same set of facts for its auxiliary predicates as the original program deduces for the original predicates.

Lemma 9.3. *Let $p \in \mathbf{idb}(\Pi)$ and $\mathbf{x}, \mathbf{y} \subseteq \mathbf{pos}(p)$ such that $p^{\mathbf{x},\mathbf{y}} \in \mathbf{idb}(\widetilde{\Pi})$. Then,*

$$\pi_{\mathbf{y}}[\Pi^{\infty}_{p}(\mathcal{D})] \subseteq \widetilde{\Pi}^{\infty}_{p^{\mathbf{x},\mathbf{y}}}(\mathcal{D}).$$

Proof. Let $f \in \Pi_p^\infty(\mathcal{D})$ be a p-fact and let $\vec{\alpha} = \alpha_1 \cdots \alpha_k$ where $\alpha_i \in \Pi$ be a sequence of rule applications which deduces f, i.e., $f \in \vec{\alpha}(\mathcal{D})$. We show a sequence $\vec{\beta}$ of the rules of $\widetilde{\Pi}$ which deduces $\pi_{\mathbf{y}} f$, i.e., a $p^{\mathbf{x},\mathbf{y}}$-fact corresponding to f. Specifically, the sequence $\vec{\beta}$ is a concatenation of k sequences as follows,

$$\vec{\beta} = \vec{\beta}_1 \cdots \vec{\beta}_k$$

where subsequence $\vec{\beta}_i$ corresponds to rule $\alpha_i \in \vec{\alpha}$. The rules in sequence $\vec{\beta}_i$ are all the rules in $\mathcal{A}(\alpha_i)$ applied in some order.

The lemma is proved by using mathematical induction on the length of $\vec{\alpha}$ to show that

$$\pi_{\mathbf{y}}[\vec{\alpha}_p(\mathcal{D})] \subseteq \vec{\beta}_{p^{\mathbf{x},\mathbf{y}}}(\mathcal{D}) \tag{9.4}$$

holds for all $p \in \mathbf{idb}(\Pi)$ and for all $p^{\mathbf{x},\mathbf{y}} \in \mathbf{idb}(\widetilde{\Pi})$.

Let $\vec{\alpha}^\ell$ denote the ℓ-lengthed prefix of $\vec{\alpha}$.

The inductive base trivially holds since if $|\vec{\alpha}| = 0$, then $|\vec{\beta}| = 0$ and therefore $\vec{\alpha}_p(\mathcal{D}) = \vec{\beta}_{p^{\mathbf{x},\mathbf{y}}}(\mathcal{D}) = \mathcal{D}$.

Let $\vec{\alpha} = \vec{\alpha}^\ell \alpha_{\ell+1}$ and let $\vec{\beta} = \vec{\beta}^\ell \vec{\beta}_{\ell+1}$ and suppose that (9.4) holds for $\vec{\alpha}^\ell$ and $\vec{\beta}^\ell$. Let $\alpha_{\ell+1}$ be "renamed" by r and let $p = \mathrm{pred}(r)$. Then (9.4) trivially holds for all $s \in \Pi$, and all $s^{\mathbf{x},\mathbf{y}} \in \widetilde{\Pi}$ such that $s \neq p$.

To show that (9.4) holds for a specific $p^{\mathbf{x},\mathbf{y}}$, consider the rule $r^{\mathbf{x},\mathbf{y}} \in \vec{\beta}_{\ell+1}$ which by definition is applied somewhere in $\vec{\beta}_{\ell+1}$.

Let $\mathcal{F}_p = \vec{\alpha}_p^\ell(\mathcal{D})$, and $\mathcal{F}'_{p^{\mathbf{x},\mathbf{y}}} = \vec{\beta}^\ell_{p^{\mathbf{x},\mathbf{y}}}(\mathcal{D})$.

Then, by the inductive hypothesis it holds that

$$\pi_{\mathbf{y}}[\mathcal{F}_p] \subseteq \mathcal{F}'_{p^{\mathbf{x},\mathbf{y}}}$$

for all $p \in \mathbf{idb}(\Pi)$ and for all $p^{\mathbf{x},\mathbf{y}} \in \mathbf{idb}(\widetilde{\Pi})$.

Let

$$\mathcal{F} = \bigcup_{p \in \mathbf{idb}(\Pi)} \mathcal{F}_p = \vec{\alpha}^\ell(\mathcal{D}), \tag{9.5}$$

and

$$\mathcal{F}' = \bigcup_{p^{\mathbf{x},\mathbf{y}} \in \mathbf{idb}(\widetilde{\Pi})} \mathcal{F}_{p^{\mathbf{x},\mathbf{y}}} = \vec{\beta}^\ell(\mathcal{D}). \tag{9.6}$$

Since $\mathcal{F}$ and $\mathcal{F}'$ satisfy the condition of Lemma 9.2, we can apply Lemma 9.2 on $\mathcal{F}$, $\mathcal{F}'$, p-rule r and $r^{\mathbf{x},\mathbf{y}}$ to obtain

$$\pi_{\mathbf{y}}[r_p(\mathcal{F})] \subseteq r^{\mathbf{x},\mathbf{y}}(\mathcal{F}').$$

Substituting (9.5) and (9.6) to the above we obtain

$$\pi_{\mathbf{y}}[r_p(\vec{\boldsymbol{\alpha}}^{\ell}(\mathcal{D}))] \subseteq r^{\mathbf{x},\mathbf{y}}(\vec{\boldsymbol{\beta}}^{\ell}(\mathcal{D})).$$

Taking into account that there is only one rule in $\vec{\boldsymbol{\beta}}_{\ell+1}$ which defines $p^{\mathbf{x},\mathbf{y}}$, we arrive to

$$\pi_{\mathbf{y}}[\vec{\boldsymbol{\alpha}}_p(\mathcal{D})] \subseteq \vec{\boldsymbol{\beta}}_{p^{\mathbf{x},\mathbf{y}}}(\mathcal{D}),$$

which completes the inductive step and ends the proof. □

Before proceeding to the next lemma, we would like to consider rules of predicates of the form $p^{\mathbf{pos}(p),\mathbf{pos}(p)} \in \mathbf{idb}(\widetilde{\Pi})$. As it turns out, each such rules can be obtained by an "extension" of the corresponding rule of Π.

More precisely,

Observation 9.8. *Let r be a p-rule, $p \in \mathbf{idb}(\Pi)$ and $\overline{r}$ be a $\overline{p}$-rule, $\overline{p} \in \mathbf{idb}(\widetilde{\Pi})$. Then, for all $a \in body(r)$, $a = s(\mathtt{x}_1, \ldots, \mathtt{x}_n)$ the following hold:*

1. *if s is an EDB predicate, then $a \in body(\overline{r})$,*
2. *otherwise, there is an atom $a' \in body(\overline{r})$ such that $a' = \overline{s}(\mathtt{x}_1, \ldots, \mathtt{x}_n)$.*

Proof. Since every predicate appearing in Π is variable-bounded, it holds that $\mathbf{terms}(\mathrm{head}(r)) \rightsquigarrow \mathbf{terms}(r)$. Now the observation follows from Definition 9.5 on construction of adapted rules. An example of such auxiliary predicate can be found in Example 9.7. □

Note however that not all atoms in $\overline{r}$ are rewrite of atoms in r. Rule $\overline{r}$ may contain many more atoms than those described by Observation 9.8. Those atoms reflect, by construction, the iterative process by which "full" atoms of the form $\overline{s}(\mathtt{x}_1, \ldots, \mathtt{x}_n)$ are evaluated.

The following lemma states that for all $p \in \mathbf{idb}(\Pi)$, a predicate p and a predicate $p^{\mathbf{pos}(p),\mathbf{pos}(p)} \in \mathbf{idb}(\widetilde{\Pi})$ are equivalent, i.e., deduce the same set of facts.

Lemma 9.4. *For all $p \in \mathbf{idb}(\Pi)$ it holds that*

$$\Pi^{\infty}_{p}(\mathcal{D}) = \widetilde{\Pi}^{\infty}_{\overline{p}}(\mathcal{D}).$$

Proof. Using Lemma 9.3 we obtain that

$$\Pi^{\infty}_{p}(\mathcal{D}) \subseteq \widetilde{\Pi}^{\infty}_{\overline{p}}(\mathcal{D})$$

holds for all $p \in \mathbf{idb}(\Pi)$. It is left to show that

$$\Pi^{\infty}_{p}(\mathcal{D}) \supseteq \widetilde{\Pi}^{\infty}_{\overline{p}}(\mathcal{D})$$

holds for all $p \in \mathbf{idb}(\Pi)$.

Let $f \in \widetilde{\Pi}^{\infty}_{\overline{p}}(\mathcal{D})$ be a $\overline{p}$-fact and let $\vec{\boldsymbol{r}} = r_1 \cdots r_k$, $r_i \in \widetilde{\Pi}$, be a sequence of rule applications which deduces f, i.e., $f \in \vec{\boldsymbol{r}}(\mathcal{D})$. We show a sequence $\vec{\boldsymbol{\tau}}$ of rules of Π which deduces f. Precisely, $\vec{\boldsymbol{\tau}} = \tau_1 \cdots \tau_k$ where $\tau_i \in \Pi$ is the corresponding original rule of r_i.

By using the same induction technique as in Lemma 9.3 we can show that

$$\pi_{\mathbf{y}}[\vec{\boldsymbol{\tau}}_p(\mathcal{D})] \subseteq \vec{\boldsymbol{r}}_{p^{\mathbf{x},\mathbf{y}}}(\mathcal{D}) \tag{9.7}$$

holds for each $p^{\mathbf{x},\mathbf{y}} \in \mathbf{idb}(\widetilde{\Pi})$ defined by a rule $r_i \in \vec{\boldsymbol{r}}$ and its original predicate $p \in \mathbf{idb}(\Pi)$.

Next we would like to refine the above statement and to show that

$$\vec{\boldsymbol{\tau}}_p(\mathcal{D}) = \vec{\boldsymbol{r}}_{\overline{p}}(\mathcal{D}) \tag{9.8}$$

holds for all $p \in \mathbf{idb}(\Pi)$.

The proof of (9.8) is done by induction on the length of $\vec{\boldsymbol{r}}$. The inductive base is when $|\vec{\boldsymbol{r}}| = 1$, i.e., $\vec{\boldsymbol{r}} = r_1$ and $\vec{\boldsymbol{\tau}} = \tau_1$. Then, if τ_1 contains only atoms of EDB predicates, so is the adapted rule r_1 (see Observation 9.8) and therefore (9.8) holds. Otherwise, τ_1 contains an atom of IDB predicate, and (9.8) trivially holds since both sequences deduce the empty set of facts.

Let $\vec{\boldsymbol{r}}^{\ell}$ denote the ℓ-lengthed prefix of $\vec{\boldsymbol{r}}$. Let $\vec{\boldsymbol{r}} = \vec{\boldsymbol{r}}^{\ell} r_{\ell+1}$ and $\vec{\boldsymbol{\tau}} = \vec{\boldsymbol{\tau}}^{\ell} \tau_{\ell+1}$ where $r_{\ell+1}$ is $\overline{p}$-rule. By the inductive hypothesis it holds that $\vec{\boldsymbol{\tau}}^{\ell}_{p}(\mathcal{D}) = \vec{\boldsymbol{r}}^{\ell}_{\overline{p}}$ for all $p \in \mathbf{idb}(\Pi)$.

Let μ be a satisfying assignment to $\mathbf{body}(r_{\ell+1})$-atoms. Then, from the inductive hypothesis and (9.7) it holds that μ is also a satisfying assignment to $\mathbf{body}(\tau_{\ell+1})$-atoms. Denoting $r_{\ell+1}$ by r' and $\tau_{\ell+1}$ by τ' we receive that for all $p \in \Pi$ it holds that

$$r'_{\overline{p}}(\vec{\boldsymbol{r}}^{\ell}(\mathcal{D})) \subseteq \tau'_{p}(\vec{\boldsymbol{\tau}}^{\ell}(\mathcal{D})),$$

which is the same as

$$\vec{\boldsymbol{r}}_{\overline{p}}(\mathcal{D}) \subseteq \vec{\boldsymbol{\tau}}_{p}(\mathcal{D}).$$

The opposite containment direction follows from (9.7). □

The following lemma states that the goal predicate set of facts of the original program is a superset of the goal predicate set of facts of the adapted program.

Let $\widetilde{q} = q^{\emptyset,\mathbf{pos}(q)}$ denote the goal predicate of $\widetilde{\Pi}$.

Lemma 9.5.

$$\widetilde{\Pi}^{\infty}_{\widetilde{q}}(\mathcal{D}) \subseteq \Pi^{\infty}_{q}(\mathcal{D}).$$

Proof. The safety property of Π states that $\emptyset \rightsquigarrow \mathbf{pos}(q)$, while the variable-bound property of Π states that $\mathsf{pos2term}_r(\mathbf{pos}(q)) \rightsquigarrow \mathbf{terms}(r)$ where $r \in \Pi$ is the rule which defines q. From these it follows that $\emptyset \rightsquigarrow \mathbf{terms}(r)$. Consider the rule $\widetilde{r} \in \widetilde{\Pi}$ which defines $\widetilde{q}$. Similarly to Observation 9.8 we can show that for all $a \in \mathbf{body}(r)$, where $a = p(\mathtt{x}_1, \ldots, \mathtt{x}_n)$, the adapted rule body $\mathbf{body}(\widetilde{r})$ contains the $p^{\mathbf{pos}(p),\mathbf{pos}(p)}(\mathtt{x}_1, \ldots, \mathtt{x}_n)$ atom. Using Lemma 9.4 we can show that each satisfying assignment μ to $body(\widetilde{r})$ atoms also satisfies $\mathbf{body}(r)$ atoms. The lemma follows. □

Theorem 9.1. *The adapted program* $\widetilde{\Pi}$ *is equivalent to* Π.

Proof. To prove the theorem it is sufficient to show that $\Pi^{\infty}_{q}(\mathcal{D}) = \widetilde{\Pi}^{\infty}_{\widetilde{q}}(\mathcal{D})$ for all $\mathcal{D} \models \mathcal{C}$. Lemma 9.3 shows that

$$\Pi^{\infty}_{q}(\mathcal{D}) \subseteq \widetilde{\Pi}^{\infty}_{\widetilde{q}}(\mathcal{D}),$$

while Lemma 9.5 shows the converse, i.e.,

$$\Pi^{\infty}_{q}(\mathcal{D}) \supseteq \widetilde{\Pi}^{\infty}_{\widetilde{q}}(\mathcal{D}).$$ □

9.2.3 Termination

In this section we prove that our algorithm terminates on safe programs with variable-bounded predicates evaluated over founded databases. In previous section we showed that the run of QSQR algorithm on the adapted program $\widetilde{\Pi}$ represents the run of our evaluation algorithm on the original program Π. After proving that Π is equivalent to $\widetilde{\Pi}$ it is enough to show that QSQR evaluation algorithm terminates on $\widetilde{\Pi}$. To show the above, we need to introduce the following lemma.

Lemma 9.6. *If* Π *is safe and all its predicates are variable bound, then* $\widetilde{\Pi}$ *is safe and all* $\widetilde{\Pi}$*'s predicates are variable bound.*

Proof. The safety property follows from the fact that $\widetilde{\Pi}$ is equivalent to Π. The variable bound property of each predicate follows from the construction. □

Thus, we need to show that the original QSQR algorithm terminates on safe program with all variable bound predicates while run over infinite and founded database. For the reminder of this section we will refer to QSQR algorithm only. As we mentioned before, in spite the fact that we do not present here the full description of the QSQR algorithm, it is easy to see that the main difference between it and our evaluation algorithm is in the way a rule evaluation is carried out.

Let us examine possible reasons for non-termination. First, the database might be accessed for infinitely many values. Second, the algorithm may recurse indefinitely. Although the algorithm takes care of this predicament (see line 5 in Algorithm 4), it is effective only when the database is finite. In the infinite case, the idb_eval function may recurse indefinitely with the same value for a p predicate and with different value for Q each time. And the last, but not least, the algorithm may not terminate if the value of M variable in Algorithm 4 never reaches the fixpoint. While the first scenario is impossible, the other two can happen if infinitely many database values are exposed during the algorithm execution.

For this sake, we show that during the run of the algorithm on safe programs defined over founded database only finitely many database values are exposed. This is precisely the case when the algorithm operates on finite database.

The following lemma rephrases the above for the general case.

Lemma 9.7. *Let* Π *be a safe program having only variable bound predicates and* $\mathcal{D}$ *be an infinite and founded database. Then, there exists a* finite *database* $\mathcal{D}' \subset \mathcal{D}$ *such that*

$$\Pi_q^\infty(\mathcal{D}) = \Pi_q^\infty(\mathcal{D}').$$

Proof. Lemma 7.5 states that $\mathbf{sources}^*(\mathcal{D})$ is finite whenever $\mathcal{D}$ is founded. The facts of $\mathcal{D}'$ are constructed as follows.

For each predicate $p \in \mathbf{edb}(\Pi)$, we create an EDB predicate s which will be part of $\mathcal{D}'$. Before presenting the facts of s, we create an auxiliary predicate s' whose facts are

given by

$$\{s'(\mathtt{x},\mathtt{y}) \mid \mathtt{x},\mathtt{y} \in \textbf{sources}^*(\mathcal{D})\}\,.$$

Then, s-facts are simply an intersection of p-facts and s'-facts and can be represented by the following rule:

$$s(\mathtt{x},\mathtt{y}) \leftarrow p(\mathtt{x},\mathtt{y}), s'(\mathtt{x},\mathtt{y}).$$

The same construction can be presented in a relational style as follows. Let

$$C = \textbf{sources}^*(\mathcal{D}) \times \textbf{sources}^*(\mathcal{D})$$

be a binary and finite relation. Then, the database $\mathcal{D}'$ contains every relation $R' = R \bowtie C$, such that $R \in \mathcal{D}$.

It is immediate to see that the lemma follows. □

Now it is left to show that during the run of the algorithm, only tuples from $\mathcal{D}'$ are retrieved.

Let $\mathbf{qsqr}_{\mathcal{D}}$ denote the set of facts retrieved from $\mathcal{D}$ during the run of QSQR algorithm. Then, we need to show that

Lemma 9.8.

$$\mathbf{qsqr}_{\mathcal{D}'} = \mathbf{qsqr}_{\mathcal{D}}.$$

Proof. It is trivial to see that

$$\mathbf{qsqr}_{\mathcal{D}'} \subseteq \mathbf{qsqr}_{\mathcal{D}}.$$

Let f be a $\mathcal{D}$-fact used by the algorithm, i.e., $f \in \mathbf{qsqr}_{\mathcal{D}}$. We will show that $f \in \mathcal{D}'$. The algorithm employs a straightforward top-down evaluation strategy, in which the primary motivation is to avoid producing facts that do not participate in derivation of any answer facts, i.e., it retrieves to the extent possible, only the required facts from the database. Consider again the value of $\textbf{consts}^*(\Pi)$ which denotes the set of all different values that can be assigned to nodes of *reduced* expansion graphs of Π. It is almost immediate to show that the set of values exposed by the top-down evaluation technique is bound above by the set of values that can be assigned to nodes of *non-reduced* expansion

graphs of Π, and by far by $\mathbf{consts}^*(\Pi)$. Therefore, from the above and from (7.2) it holds that

$$\mathbf{consts}(f) \subseteq \mathbf{consts}^*(\Pi) \subseteq \mathbf{sources}^*(\mathcal{D}).$$

The lemma follows. □

Chapter 10

Conclusion

In this research we studied the weak safety and termination problems (and thereby, also the safety problem) for recursive DATALOG programs over infinite databases. We presented an algorithm that computes all constraints for IDB predicates that are (finitely) implied by the constraints on the EDB predicates and the rules of a given program. We also showed that weak safety guarantees termination if the database is founded, a natural property in many models. Finally, for safe programs we presented an elegant evaluation algorithm that computes the goal predicate in a top-down manner, using sideways information passing.

Bibliography

[1] The Hibernate Project. `http://www.hibernate.org/`.

[2] W. W. Armstrong. Dependency structures of data base relationships. In *IFIP Congress*, pages 580–583, 1974.

[3] C. Beeri and P. A. Bernstein. Computational problems related to the design of normal form relational schemas. *ACM Trans. Database Syst.*, 4(1):30–59, 1979.

[4] S. Ceri, G. Gottlob, and L. Tanca. *Logic programming and databases*. Springer Verlag, New York, 1990.

[5] T. Cohen, J. Y. Gil, and I. Maman. JTL—the Java tools language. In P. L. Tarr and W. R. Cook, editors, *Proc. of the Twenty First Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06)*, Portland, Oregon, Oct.22-26 2006. ACM SIGPLAN Notices.

[6] R. F. Crew. ASTLOG: A language for examining abstract syntax trees. In S. Kamin, editor, *Proc. of the First USENIX Conference Domain Specific Languages (DSL'97)*, pages 229–242, Santa Barbara, Oct. 1997.

[7] S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems—a case study. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI'96)*, pages 117–126, New York, NY, USA, 1996. ACM Press.

[8] A. Deutsch, B. Ludäscher, and A. Nash. Rewriting queries using views with access patterns under integrity constraints. *Theoretical Comp. Sci.*, 371(3):200–226, 2007.

[9] D. Florescu, A. Y. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *SIGMOD 1999*, pages 311–322, New York, NY, USA, 1999. ACM Press.

[10] E. Hajiyev, M. Verbaere, and O. de Moor. CodeQuest: Scalable source code queries with Datalog. In D. Thomas, editor, *Proc. of the Twentieth European Conference on Object-Oriented Programming (ECOOP'05)*, volume 4067 of *Lecture Notes in Computer Science*, Nantes, France, July 3–7 2006. Springer Verlag.

[11] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *Proc. of the Second international conference on Aspect-Oriented Software Development (AOSD'03)*, pages 178–187, New York, NY, USA, 2003. ACM Press.

[12] S. Javey, K. Mitsui, H. Nakamura, T. Ohira, K. Yasuda, K. Kuse, T. Kamimura, and R. Helm. Architecture of the XL C++ browser. In *Proc. of the Conference of the Centre for Advanced Studies on Collaborative research (CASCON'92)*, pages 369–379, Toronto, Ontario, Canada, 1992. IBM Press.

[13] M. Kifer. On the decidability and axiomatization of query finiteness in deductive databases. *J. ACM*, 45(4):588–633, 1998.

[14] M. Kifer, R. Ramakrishnan, and A. Silberschatz. An axiomatic approach to deciding query safety in deductive databases. In *Proc. of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'88)*, Austin, Texas, Mar. 1988. ACM Press, New York, NY, USA.

[15] R. Krishnamurthy, R. Ramakrishnan, and O. Shmueli. A framework for testing safety and effective computability of extended datalog. In *Proc. of the ACM SIGMOD International Conference on Management of Data (ICMS'88)*, Chicago, Illinois, June 1988. ACM Press, New York, NY, USA.

[16] C. Li and E. Y. Chang. On answering queries in the presence of limited access patterns. In J. V. den Bussche and V. Vianu, editors, *ICDT*, volume 1973 of *Lecture Notes in Computer Science*, pages 219–233. Springer, 2001.

[17] D. Maier, J. D. Ullman, and M. Y. Vardi. On the foundations of the universal relation model. *ACM Trans. Database Syst.*, 9(2):283–308, 1984.

[18] R. Manevich. Data structures and algorithms for efficient shape analysis. Master's thesis, Tel-Aviv University, School of Computer Science, Jan. 2003.

[19] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In A. P. Black, editor, *Proc. of the Nineteenth European Conference on Object-Oriented Programming (ECOOP'05)*, volume 3086 of *Lecture Notes in Computer Science*, pages 214–240, Glasgow, UK, July 25–29 2005. Springer Verlag.

[20] R. D. Paola. The recursive unsolvability of the decision problem for the class of definite formulas. *J. ACM*, 16(2):324–327, 1969.

[21] R. Ramakrishnan, F. Bancilhon, and A. Silberschatz. Safety of recursive Horn clauses with infinite relations. In *Proc. of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'87)*, San Diego, California, Mar. 1987. ACM Press, New York, NY, USA.

[22] T. Reps. Shape analysis as a generalized path problem. In *PEPM '95: Proc. of the 1995 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 1–11, New York, NY, USA, 1995. ACM Press.

[23] Y. Sagiv and M. Y. Vardi. Safety of Datalog queries over infinite databases. In *Proc. of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'89)*, pages 160–171, Philadelphia, Pennsylvania, United States, Mar. 1989. ACM Press, New York, NY, USA.

[24] O. Shmueli. Decidability and expressiveness aspects of logic queries. In *Proc. of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'87)*, San Diego, California, Mar. 1987. ACM Press, New York, NY, USA.

[25] M. Vardi. The decision problem for database dependencies. *Inf. Process. Lett.*, 12(5):251–254, 1981.

[26] L. Vieille. Recursive axioms in deductive databases: The Query/Subquery approach. In L. Kerschberg, editor, *Proc. of the FirstInt. Conf. on Expert Database Syst.*, pages 179–197, Redwood City, CA, USA, 1986. Benjamin-Cummings Publishing Co., Inc.

[27] L. Vieille. Recursive query processing: The power of logic. *Theoretical Comp. Sci.*, 69(1):1–53, 1989.

[28] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using Datalog and binary decision diagrams for program analysis. In K. Yi, editor, *Proc. of the Third Asian Symposium on Programming Languages and Systems*, volume 3780 of *Lecture Notes in Computer Science*. Springer Verlag, Nov. 2005.

[29] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI'04)*, pages 131–144, New York, NY, USA, June 9-11 2004. ACM Press.

$\{dep\_parents_2\} \rightsquigarrow \{dep\_parents_1\}$ משום שהן בפרט גם עובדות על פרדיקט parents. כעת

פרדיקט המטרה q מכיל רק את העובדות על dep_parents המתייחסות ל-'Bill' במיקום הראשון.
כיוון שהמיקום הראשון קובע באופן סופי את המיקום השני, ניתן להסיק כי ישנו רק מספר סופי של עובדות על q.)
הסיבה בגללה לא ניתן לשערך את התוכנית הנתונה היא הקושי לשערך את הפרדיקט parents. אנחנו איננו יכולים אף לבדוק האם רשומה המתייחסת לפרדיקט parents אכן מהווה עובדה, כלומר נכונה.
במשפט 8.1. אנחנו קובעים מתי התוכנית ניתנת לשיערוך במודל החישוב שלנו. עבור תוכניות כאלה פיתחנו אלגוריתם שיערוך מתאים. האלגוריתם עובד בשיטת top-down ומבוסס על אלגוריתם שיערוך QSQR של Vieille. התזה מסתיימת בהוכחת נכונות האלגוריתם שלנו.

מחקר עתידי

כהמשך אפשרי של העבודה שלנו ניתן לבדוק את הבעיות הנידונות בעבודה במודל של Datalog המתיר שלילה.

בעוד שבעיית הסיום של תוכניות Datalog מעל מסדי נתונים אינסופיים הוכחה כבלתי כריעה על ידי שגיב וורדי, אני מוכיחה במשפט 7.1. כי כאשר מסד הנתונים גם מבוסס, כל תוכנית אשר בטוחה באופן חלש הינה גם עוצרת ולכן גם בטוחה.

שערוך

אחרי שיישבנו את שאלת הבטיחות, ברצוננו לעסוק בבעיה של שיערוך תוכניות בטוחות. כיוון שמדובר במסד נתונים אינסופי, עלינו להגדיר מודל חישוב מדויק.
במודל החישוב שלנו אילוצי הסופיות על פרדיקטים התחומיים קובעים את השאילתות המותרות על מסד הנתונים.
בפרט, אם p הינו פרדיקט EDB (בינארי לפי ההנחה), אזי רק השאילתות הבאות מותרות ומתבצעות בזמן סופי:

1. בהינתן שני קבועים c ו-'c, ניתן לקבוע האם p(c, 'c) מהווה עובדה.
2. אם ידוע כי $\{p_2\} \rightsquigarrow \{p_1\}$ ובהינתן הקבוע c, ניתן למצוא את כל הקבועים 'c כך ש-p(c, 'c) מתקיים. באופן דומה לגבי האילוץ ההפוך $\{p_2\} \rightsquigarrow \{p_1\}$.
3. אם ידוע האילוץ $\varnothing \rightsquigarrow \{p_1\}$, אז ניתן למצוא את כל הקבועים c כך שקיים הקבוע 'c עבורו מתקיים ש-p(c, 'c) הינה עובדה. באופן דומה לגבי האילוץ $\varnothing \rightsquigarrow \{p_2\}$.

בתחילה הבעיה של היכולת לשערך נראתה כטריוויאלית. כלומר, סברנו כי מספיק שהתוכנית תהיה בטוחה בכדי שנוכל לשערך אותה תחת מודל החישוב הנ"ל. בסופו של יום, הסתבר שלא כך הדבר.
להלן תוכנית עם פרדיקט מטרה q אשר ממחישה זאת:

parents(x, y) ← child(w, x), child(z, y).

dep_parents(x, y) ← dependant(x, y), parents(x, y).

q(y) ← dep_parents('Bill', y).

תוכנית זאת משתמשת בפרדיקטים תחומיים child ו-dependant. בפרט, child('c', 'p') מתקיים אם 'c' הינו הילד של 'p' ו-dependant('d1', 'd2') מתקיים אם 'd1' תלוי ב-'d2'. קל לראות כי כאשר האילוצים הבאים מתקיימים

$\{child_1\} \rightsquigarrow \{child_2\}, \{dependant_1\} \rightsquigarrow \{dependant_2\}$

ומסד הנתונים הינו מבוסס, התוכנית הינה בטוחה כי האילוץ $\varnothing \rightsquigarrow \{q_1\}$ מתקיים. (אמנם לא הצגנו במסגרת התקציר את האלגוריתם לחישוב אילוצים על הפרדיקטים הכוונתיים, אך כיוון שתוכנית זאת אינה רקורסיבית אין זה נחוץ. בפרט, עובדות על פרדיקט parents, הנוצרים על ידי הפעלה אחת של החוק הראשון, אינם מקיימים אף אילוץ סופיות. עובדות על dep_parents מקיימות אילוץ יחיד

הגבלה כזאת מבוטאת באמצעות אילוצי סופיות. בפרט האילוץ עבור פרדיקט child מוצג בדרך הבאה:

$$\{child_1\} \rightsquigarrow \{child_2\} \quad (1)$$

כאשר $child_i$ מסמן מיקום (position) ה-i-י בפרדיקט child.

נרצה להתייחס רק למסדי נתונים בהם אוסף כל העובדות על פרדיקט child מספק את האילוץ הנ"ל, כלומר עבור כל קבוע 'c' במיקום ראשון קיים מספר סופי של ערכי 'p' כך ש-child('c', 'p') מתקיים.

העבודה שלנו מתבססת על עבודתם של יהושע שגיב ומשה ורדי אשר דנה בבטיחות של תוכניות Datalog מעל מסדי נתונים אינסופיים המקיימים אילוצי סופיות. שגיב וורדי מחלקים את שאלת הבטיחות לשתי תתי שאלות:

1. בעיית בטיחות חלשה (weak safety). התוכנית תקרא **בטוחה באופן חלש** אם כל מספר סופי של הפעלות של חוקי התוכנית יניבו מספר סופי של עובדות לפרדיקט המטרה.
2. בעיית הסיום (termination). התוכנית תקרא **מסתיימת** אם ניתן לקבל את כל העובדות של פרדיקט המטרה על ידי מספר סופי של הפעלות של חוקי התוכנית.

שני החוקרים הוכיחו כי בעיית הבטיחות החלשה ניתנת להכרעה. לעומת זאת מסתבר כי בעיית הסיום אינה כריעה.

עבודה זו מציגה פתרון שונה לבעיית הבטיחות החלשה. כהתחלה, אנחנו מציגים אלגוריתם אשר בהינתן אילוצי סופיות על הפרדיקטים התחומיים , מחשב את כל האילוצים על הפרדיקטים הכוונתיים.

לדוגמא, עבור התוכנית אשר מחשבת אבות קדמונים של 'Moses' ומוגדרת מעל פרדיקט child עליו ידוע האילוץ (1):

moses_ancestor(x) ← descendant('Moses', x).

descendant(x, y) ← child(x, z), descendant (z, y).

descendant (x, y) ← child(x, y).

האלגוריתם יסיק כי האילוצים $\{descendant_1\} \rightsquigarrow \{descendant_2\}$ ו- $\varnothing \rightsquigarrow \{moses\_ancestor_1\}$ מתקיימים.

כעת, בהינתן תוצאות הריצה של האלגוריתם נוכל להכריע את בעיית הבטיחות החלשה. אנחנו מראים במשפט 6.1. כי תוכנית היא בטוחה באופן חלש אמ"מ עבור פרדיקט המטרה של התוכנית q בעל arity בגודל k מתקיים האילוץ $\varnothing \rightsquigarrow \{q_1,\ldots,q_k\}$.

מכאן נוכל להסיק כי התוכנית המחשבת את האבות הקדמונים של 'Moses' הינה בטוחה באופן חלש. על מנת להראות שתוכנית זו גם בטוחה, יש להוכיח כי התוכנית גם מסתיימת, כלומר די במספר סופי של הפעלות של חוקי התוכנית על מנת להסיק את כל העובדות על פרדיקט המטרה.

```
public [extends T, T abstract | interface];
```

תוכנית ה-Datalog השקולה לה היא:

$q(x) \leftarrow public(x), p(x).$

$p(x) \leftarrow interface(x).$

$p(x) \leftarrow extends(x, y), abstract(y).$

בתוכנית זאת משתמשים בארבעה פרדיקטים **תחומיים** (Extensional DataBase או בקיצור EDB) – public, interface, extends ו-abstract. פרדיקטים אלו מייצגים רלציות הנמצאות במסד הנתונים. כמו כן מוגדרים בתוכנית שני פרדיקטים **כוונתיים** (Intensional DataBase או בקיצור IDB) – q ו-p. אלו הם פרדיקטים המתאימים לרלציות שאינן נמצאות במסד הנתונים ותוכנן מוגדר ע״י חישוב התוכנית. שורה בתוכן של רלציה תיקרא גם **עובדה** עבור הפרדיקט המתאים.
בתוכנית זאת q הינו **פרדיקט המטרה** ועניינו מסתכם במציאת עובדות התואמות לפרדיקט זה.
ב-JTL ניתן גם להביע שאילתות אשר יתורגמו לתוכניות Datalog רקורסיביות.

כנאמר לעיל, גם בעולם התוכנה המיוצג על ידי מסד נתונים ובו רלציות אינסופיות ישנם אילוצי סופיות מובנים. כך למשל, רק מספר סופי של מחלקות יכול לרשת ממחלקה נתונה. אך קיימת עוד אינדיקציה של סופיות במסד נתונים כזה. נתבונן למשל ביחס בינארי "uses" אשר מבטא קשר שימוש כלשהו בין שתי מחלקות, כגון ירושה, קריאה למתודה וכו׳. הסגור הטרנזיטיבי של יחס זה הינו חסום במובן הבא: אמנם מספר המחלקות המשתמשות במחלקה מסוימת לא ידוע ויכול להיות לא חסום, אך מספר המחלקות אשר המחלקה הנוכחית משתמשת בהן, בין אם באופן ישיר או עקיף, הוא סופי. לתכונה הזאת של מסדי נתונים אנחנו קוראים מסדי נתונים **מבוססים**. בנוסף נניח (למען הפשטות) כי כל הפרדיקטים התחומיים הינם בינאריים.

בטיחות

מרכז התעניינותנו על כן הוא בתוכניות Datalog המופעלות מעל מסדי נתונים אינסופיים ומבוססים. כצעד ראשון, נרצה לקבוע עבור תוכניות כאלה האם הן בטוחות.
נתבונן בדוגמא פשוטה המוגדרת עם פרדיקט תחומי child:

$moses\_parent(x) \leftarrow child('Moses', x).$

הפתרון של התוכנית יניב עובדות לפרדיקט moses_son אשר יכילו את שמות כל ההורים של 'Moses'. אם מסד הנתונים אינו מוגבל כלל, התוצאה עלולה להיות אינסופית. אך, אם נקבע שבכל מסד נתונים אינסופי לכל ילד יכולים להיות מספר סופי של הורים, התוצאה תמיד תהיה סופית ולכן נסיק כי התוכנית בטוחה.

תקציר

בדרך כלל מסדי נתונים מכילים רלציות סופיות, אך ישנם מצבים אשר המידול הטוב ביותר שלהם מוביל למסדי נתונים אינסופיים. לרוב, במקרים כאלו מוגדרים על מסדי הנתונים **אילוצי סופיות**, אשר עוזרים לבטא הגבלות על תכנים של רלציות אינסופיות.

המחקר שלנו עוסק בשפת שאילתא לוגית Datalog אשר מיושמת מעל מסדי נתונים אינסופיים כאלו.

הבעיה העיקרית בה אני עוסקת היא השאלה האם תוכנית Datalog הינה **בטוחה**, כלומר האם היא מניבה תמיד תוצאה סופית על אף שמסד הנתונים הוא אינסופי. בנוסף, עבור תוכניות בטוחות ברצוננו להציג אלגוריתם שערוך הרץ בזמן סופי אשר מוצא את התוצאות.

שימוש קלאסי במסדי נתונים אינסופיים הוא להרחבה של Datalog לטיפול בסימני פונקציות. באופן זה, נוח לעבור לתוכנית Datalog רגילה ואילו לייצג פונקציות על ידי רלציות אינסופיות.

לדוגמא, התוכנית המכילה שני סימני פונקציה

$q(x + 1) \leftarrow p(x).$

$q(x) \leftarrow q(\sqrt{x}), p(x).$

ניתנת לייצוג על ידי תוכנית Datalog הבאה:

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

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

כעת התוכנית מוגדרת ללא סימני פונקציה ואילו מסד הנתונים מכיל שתי רלציות אינסופיות sqrt ו-succ. על מנת לקרב את התוכן של הרלציות האינסופיות למודל פונקציות, מוסיפים אילוצי סופיות.כך למשל ברלציה sqrt נשתמש באילוץ הסופיות שלכל x קיים מספר סופי של y-ים כך ש-sqrt(x, y) מתקיים.

מוטיבציה

עניין מחודש במסדי נתונים אינסופיים נוצר לאחרונה עם הניסיון לייצג בצורה פורמאלית את עולם התוכנה או את אוסף הרלציות אשר פזורות ב-web. עולם התוכנה הדינאמי הינו אינסופי במובן שהוא אינו פוסק מלגדול ועל כן לא ניתן לסרוק אותו במלואו באף רגע נתון. לדוגמא, בהינתן מחלקה C יתכן מספר לא חסום של מחלקות אשר יורשות ממנה, אשר קוראות למתודה שלה או בעלות שדה המצביע ל-C. על כן, הדרך הטבעית ביותר הממדלת הפעלת שאילתות בעולם התוכנה היא כשמסד הנתונים הינו אינסופי.

המוטיבציה שלנו למחקר הייתה שפת שאילתות JTL אשר מאפשרת לשאול שאילתות על תוכנה. השפה מבוססת על Datalog. ב-JTL מודל הנתונים מיוצג על ידי מסד נתונים אינסופי.

לדוגמא שאילתת JTL אשר מוצאת את (1) כל המנשקים הפומביים (public interfaces) או (2) את כל המנשקים או מחלקות פומביים אשר יורשים ממחלקה או מנשק אבסטרקטי.

רשימת אלגוריתמים

רשימת תרשימים

תוכן העניינים

המחקר נעשה בהנחייתו של ד״ר יוסי גיל
בפקולטה למדעי המחשב

אני מודה לטכניון – מכון טכנולוגי לישראל על התמיכה הכספית
הנדיבה בהשתלמותי

שיערוך תוכניות Datalog מעל מסדי נתונים אינסופיים ומבוססים

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים במדעי המחשב

אבלינה זריבץ'

הוגש לסנט הטכניון – מכון טכנולוגי לישראל
טבת תשס"ח חיפה דצמבר 2007

שיערוך תוכניות Datalog מעל מסדי נתונים אינסופיים ומבוססים

אבלינה זריבץ'