National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services Branch

Direction des acquisitions et des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file   Votre référence

Our file   Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

EFFICIENT EVALUATION OF FUNCTIONAL RECURSIVE QUERY PROGRAMS IN DEDUCTIVE DATABASES

by

Qiang Wang

B.Sc., Beijing Computer Institute, 1984

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the School

of

Computing Science

© Qiang Wang 1991

SIMON FRASER UNIVERSITY

January 1991

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without permission of the author.

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services Branch

Direction des acquisitions et des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-78309-5

Canada

Approval

Name: Qiang Wang

Degree: Master of Science

Title of thesis: Efficient Evaluation of Functional Recursive Query Programs in Deductive Databases

Examining Committee:

Dr. Ze-Nian Li
Chairman

Dr. Jiawei Han
Senior Supervisor

Dr. Wo-Shun Luk
Supervisor

Dr. Nick J. Cercone
Supervisor

Dr. Fred Popowich
External Examiner

January 16, 1991
Date approved

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

Efficient Evaluation of Functional Recursive Query Programs in Deductive Databases.

Author: ____________________
(signature)

Qiang Wang
(name)

January 17, 1991.
(date)

Abstract

Functional recursive query programs are recursive query programs expressed in Horn clause logic with function symbols. The functional recursions studied in this thesis are confined to one commonly used class of recursions: linear recursions.

We compile a functional linear recursion into a highly regular *compiled formula* and analyze the safety and the evaluation of the compiled formula with respect to a given query and a set of constraints specified over a given database instance. We show that for the functional linear recursions, safety can be viewed as a combination of two properties: finite evaluability, which guarantees the finiteness of intermediate answers, and termination, which guarantees the finiteness of the evaluation. We present a necessary and sufficient condition guaranteeing the finite evaluability of compiled formulas and a sufficient condition guaranteeing the termination of the evaluation. The algorithms for testing these conditions are developed.

Based on the analysis of safety, we present a safe, constraint-based evaluation method for compiled formulas. We classify constraints into three classes: (i) integrity constraints, (ii) rule constraints and (iii) query constraints. We show that integrity constraints should be used to generate safe evaluation plans. Rule constraints can be used in compilation to reduce the search space. Query constraints are shown to be useful in the selection of efficient evaluation plans and search space reduction.

Dedication

To my parents and my wife.

Acknowledgments

I feel very grateful for the patience, assistance and supervision of my senior supervisor, Dr. Jiawei Han, who has always been available when I need assistance and who provided careful and conscientious reading and corrections of the early versions of this thesis. Nobody could have provided more support, assistance and guidance for this research effort. He has also been a constant source of inspiration, without which this thesis would not have been possible.

I would like to express my thanks to Dr. Nick J. Cercone for his support throughout my thesis research and express my thanks to Dr. Wo-Shun Luk for leading me through database fields at very beginning and for helping me to be re-united with my wife. I would like to express my appreciation and gratitude to Dr. Fred Popowich for his patience and careful reading of my thesis. His comments on my thesis were very helpful.

I would also like to thank Sanjeev Mahanjan for some useful discussions.

I am thankful to Dr. Ramesh Krishnamurti, Dr. Ze-Nian Li, Russ Tront and Kersti Jagger, who made my stay at S.F.U. memorable.

A special note of appreciation to my friend, Jean Rowe, who has been always helpful from the time she met me at the Vancouver International Airport to the final stage of my thesis. Her help with the revision of this thesis and preparation for my thesis seminar was invaluable.

Lastly, to my wife Yan Tang who put up with this for more than two years and who made this possible.

Table of Contents

Approval ii

Abstract iii

Dedication iv

Acknowledgments v

List of Tables viii

List of Figures ix

1. Introduction 1

1.1 Thesis Research Area 1

1.2 An Example from Real World Applications 2

1.3 Thesis Contribution and Organization 8

2. Preliminaries 11

2.1 Deductive Databases 11

2.2 Functions in Deductive Databases 21

3. Transformation of Functional Recursions and Compiled Formulas 26

3.1 Transformation of Functional Recursions 28

3.2 Rectification of Function-Free Recursions 31

3.3 Compiled Formula Assumption 33

4. Safety of the Evaluation of Compiled Formulas 43

4.1 The Concept of Safety 44

4.2 Classification of Constraints 46

4.3 Safety as a Combination of Finite Evaluability and Termination 55

4.4 Safety I: Finite Evaluability 56

4.4.1 Testing of Finite Evaluability 58

4.4.2 Analysis of the Finite Evaluability Testing Algorithm 62

4.5 Safety II: Termination 63

4.5.1 Termination Detection 68

4.5.2 Analysis of the Termination Detection Algorithm 70

5. Constraint-Based Evaluation of Compiled Formulas 73

5.1 Incorporation of Integrity Constraints 77

5.1.1 Incorporation of Type Constraints 77

5.1.2 Incorporation of Finiteness Constraints 80

5.1.3 Incorporation of Monotonicity Constraints 86

5.2 Incorporation of Rule Constraints 89

5.3 Incorporation of Query Constraints 91

5.4 Generation of Safe, Constraint-Based Evaluation Plans 94

6. Conclusion 102

References 104

List of Tables

Table 1-1 A daily flight information table....4

Table 2-1 Some commonly used functions....23

Table 4-1 Type constraints for the EDB predicate *flight*....53

Table 5-1 The evaluation order generated for the *append* program....83

Table 5-2 The evaluation order generated for the *gcd* program....85

Table 5-3 The evaluation order generated for the *travel* program....99

List of Figures

Figure 1-1 Classification of Horn clauses. 2

Figure 1-2 Travel map. 3

Figure 1-3 Thesis organization 10

Figure 2-1 Datalog and extended Datalog 14

Figure 2-2 Deductive databases. 18

Figure 2-3 Illustration of the *travel* program. 19

Figure 2-4 Illustration of computing the area of a convex polygon 25

Figure 3-1 Transformation of functional recursions 26

Figure 3-2 Rectification of EDPs. 27

Figure 3-3 Compiled formula assumption. 27

Figure 3-4 Shared variables between chain elements 37

Figure 3-5 Chain invariant of $cons^i(X_i, U_i, U_{i-1})$. 38

Figure 4-1 Different meanings of query safety 44

Figure 4-2 Application of Algorithm 4-1 in the *plus* program 63

Figure 4-3 Two terminating evaluation plans for the *times* program 65

Figure 4-4 A finitely evaluable and terminating plan for the *mod* program. 69

Figure 5-1 Constraints used in the evaluation of compiled formulas. 73

Figure 5-2 Search space reduction by using rule constraints and query constraints. 74

Figure 5-3 Illustration of evaluation plans and evaluation orders. 75

Figure 5-4 Illustration of the evaluation of a compiled formula. 76

Figure 5-5 The finiteness propagation in the *append* program. 81

Figure 5-6 The finiteness propagation in the *gcd* program. 85

Figure 5-7 Rule constraints added on a nondistinguished variable. 91

Figure 5-8 Selection of evaluation directions according to selectivity. 95

Figure 5-9 The finiteness propagation in the *travel* program. 98

Chapter 1

Introduction

1.1 Thesis Research Area

Functional recursive query programs are recursive query programs expressed in Horn clause logic with function symbols. Their development can be viewed from two perspectives: one related to the integration of the logic and the functional programming languages [BeLe86], and the other as an effort to extend the expressive power of conventional database query languages [CGKNTZ90] [MNSUV87].

The functional recursions studied in this thesis are confined to one commonly used class of recursions: linear recursions, and we will discuss the evaluation of functional recursive query programs within the database context.

The dominant database products in today's marketplace are relational database systems, such as Ingres, Oracle, Sybase, DB2 etc. However, relational databases have two limitations. One is the limitation of their query languages, e.g., recursive queries cannot be handled effectively; the other is that the fast access capability of the data manipulation language (DML) and the general-purpose capability of the host language do not mesh well.

Deductive databases have emerged over the past few years to become one of the new trends in database research. They are mainly based on the success of relational databases and are aimed at extending the expressive power of relational databases and providing a single declarative language to serve the roles played by both the DML and the host language. Their origin and objectives suggest a promising future provided their efficiency can be reasonably guaranteed

[Ullm89b] [Date90]. A major step in achieving efficiency is to reduce the overhead of recursive query processing, viewed as one of the most costly supports of deductive databases.

One of the essential relationships between relational databases and deductive databases lies in the relationship between relational algebra and Horn clause logic. It is known that when recursions are not permitted, relational algebra is equivalent to Datalog logic (i.e., function-free Horn clause logic) in terms of expressive power [GMN84] [Ullm89b]. One of the most notable features of deductive databases is the ability to handle both function-free and functional recursions.

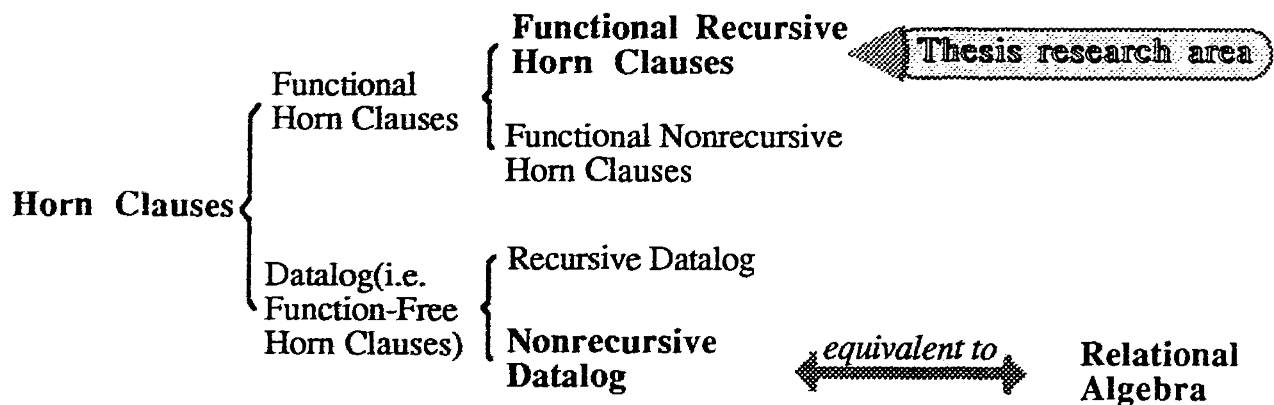


Figure 1-1 Classification of Horn clauses.

Theoretically, it is known that logic rules using function symbols have all the power of a Turing machine, i.e., they can express any computation that can be written in conventional programming languages [Ullm89b]. The example in Section 1.3 below shows the need for functional recursions in real world applications.

1.2 An Example from Real World Applications

Example 1-1: Suppose a travel agency maintains the flight information in a table called *flight* (see Figure 1-2 and Table 1-1), in which one tuple represents one flight every day. For

simplicity, we assume that the time considered in this example refers to Greenwich Mean Time[1] and the travel time for a continuing trip is not greater than twenty-four hours.

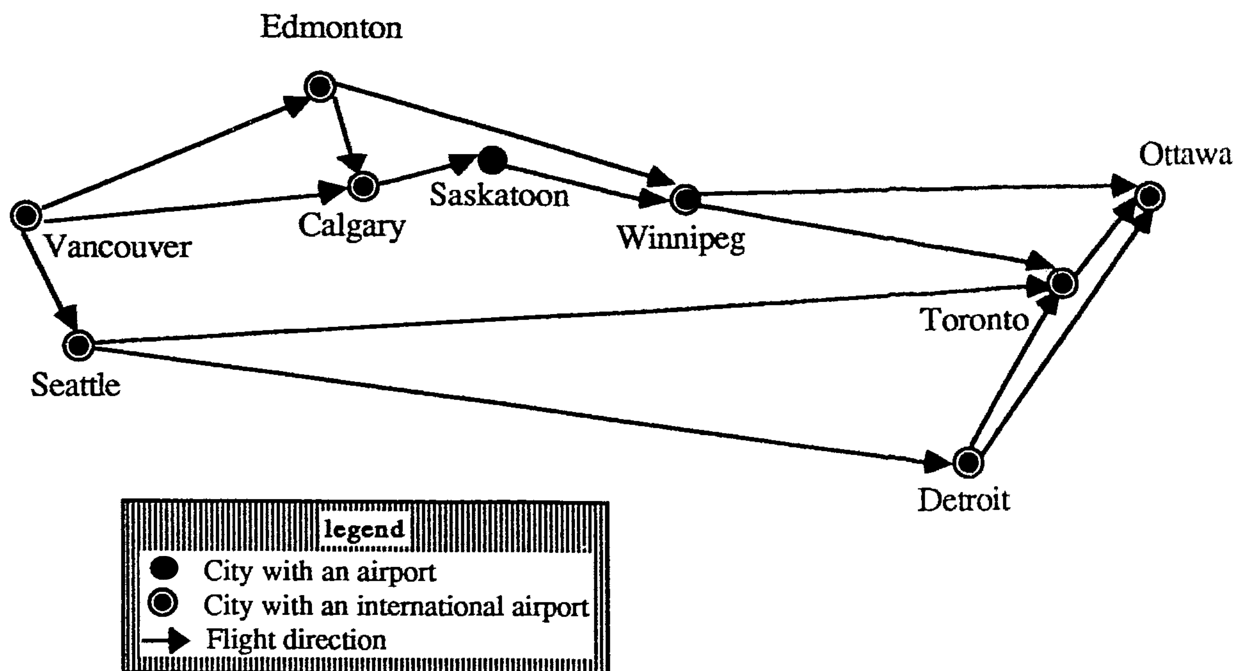


Figure 1-2 Travel map.

---

[1] In a real system Greenwich Mean Time can be converted to local time by adding the corresponding offset.

Flight Number	Departure City	Departure Time	Arrival City	Arrival Time	Fare
1	Vancouver	7:00	Edmonton	9:00	$120
2	Vancouver	7:30	Calgary	9:15	$100
3	Vancouver	23:00	Seattle	24:00	$60
4	Edmonton	8:00	Winnipeg	11:30	$200
5	Edmonton	11:00	Calgary	11:45	$50
6	Calgary	16:00	Saskatoon	17:00	$60
7	Seattle	4:00	Toronto	11:00	$500
8	Seattle	2:00	Detroit	8:00	$450
9	Saskatoon	5:00	Winnipeg	7:30	$140
10	Winnipeg	19:00	Toronto	23:30	$250
11	Winnipeg	15:00	Ottawa	20:00	$300
12	Detroit	10:00	Toronto	11:30	$90
13	Detroit	9:00	Ottawa	12:00	$130
14	Toronto	7:00	Ottawa	8:00	$70

Table 1-1 A daily flight information table

Suppose a customer arrives at the front desk with the following questions.

Question 1:

"Could you help me find a sequence of connecting flights from Vancouver to Ottawa?"

Question 2:

"I'd like to leave Vancouver between 7am and 8am for Ottawa. Could you arrange that for me?"

Question 3:

"I have an appointment in Ottawa at 2pm and I want to have lunch before the meeting. Can I get a flight, or connecting flights from Vancouver and arrive in Ottawa around noon?"

Question 4:

"My boss can only reimburse me $500 for my travel from Vancouver to Ottawa. Can I get a ticket for under $500?"

Question 5:

"I want to go from Vancouver to Ottawa. I want all intermediate stops to be at international airports. Please arrange that for me."

We analyze these questions in terms of the following three aspects: the incorporation of functions in recursive query processing, the safety of functional recursions, and the generation of efficient evaluation plans.

1. A natural and easy way to deal with these questions is to incorporate functions into the recursive query processing.

We notice that every question has at least one minimal requirement, that is, to find the flight number(s) from *Vancouver* to *Ottawa*.

Since there is no direct flight from *Vancouver* to *Ottawa*, we need use some recursive query processing techniques and introduce a function which can concatenate the connecting flight numbers (i.e., the first attribute of the relation *flight*). For example, one solution is from *Vancouver* to *Seattle*, then to *Detroit*, to *Toronto*, and finally to *Ottawa*. This solution can be represented by a list of flight numbers, that is, *[3,8,12,14]*. If we use $L = [Fno/L_1]$ to denote the list construction function which concatenates the head element *Fno* (flight number) with the list L_1 (a list of flight numbers) to form a new list *L*, the process of finding this solution can be illustrated as follows: $L = [3/L_1]$, $L_1 = [8/L_2]$, $L_2 = [12/14]$.

In addition, to give a satisfactory answer to question 4, we need another function which can sum up the flight fares during recursive query processing. If we use function $Fare = S + Fare_1$ to sum up fare *S* for the first leg of a flight and fare $Fare_1$ for the other legs of the flight, the total fare of the connecting flights *[3,8,12,14]* can be calculated as follows. $Fare = 60 + Fare_1$, $Fare_1 = 450 + Fare_2$, $Fare_2 = 90 + 70$. The total fare is *670* dollars (note that this result does not meet the "under *$500*" requirement given in question 4).

2. The evaluation must be *safe*, that is, it should generate finite intermediate and final answers in finite steps.

From the relational algebra point of view, the basic idea of recursive query processing is to perform the relational operations (particularly, the join and union operations) between relations. However, since functions are normally defined on infinite domains, the recursive query evaluation may generate infinite answers or end up with infinite derivation of answers. We discuss the two possibilities below.

Take the functions $L = [3/L_1]$ and $Fare = 60 + Fare_1$ as an example. We notice that when L_1 and $Fare_1$ are unknown, the answers derived from these two functions are essentially infinite.

Therefore, how to guarantee the finiteness of answers is crucial to the evaluation of functional recursive query programs.

Not only must the answers be finite, but the derivation of the answers must terminate in finite steps as well. For instance, suppose there is another direct flight from *Detroit* to *Vancouver*. A loop, *Vancouver* → *Seattle* → *Detroit* → *Vancouver*, is formed. This may cause an infinite derivation of answers (note that the natural join operation will be performed infinitely in this case).

3. <u>Efficient evaluation plans should be generated based on the query and the constraints (requirements) provided by the customer</u>.

Take the solution *[3,8,12,14]* with the total fare *670* dollars as an example. Clearly, this is not a satisfactory answer to question 4. If we use the "*under $500*" requirement as a constraint and push it into the evaluation, this solution will not be generated because the fare from *Vancouver* to *Detroit* is *510* dollars which already violates the constraint. Similarly, to answer question 5 which requires all the intermediate airports be international airports, an efficient evaluation plan is to consider only those flights which stop at international airports.

Take questions 2 and 3 as another example. Our intuition tells us that to answer question 2, an efficient evaluation plan is to search for answers from *Vancouver* instead of from *Ottawa*. However, this evaluation plan is not efficient for question 3 because there are 9 possibilities all together but only one of them (i.e., *Vancouver* → *Seattle* → *Detroit* → *Ottawa*) meets the requirement, "*arriving in Ottawa around noon*". An efficient evaluation plan for question 3 is to start searching from *Ottawa*.

This thesis studies the evaluation of functional recursions with respect to the above three aspects, that is, the incorporation of functions in recursive query processing, the *safety* (i.e., *finite evaluability* and *termination*) of functional recursions, and the generation of efficient evaluation plans.

1.3 Thesis Contribution and Organization

Due to the importance of recursions in deductive databases, there has been increased research into techniques for implementing recursions in recent years. This research, however, has not converged into one widely accepted approach, but instead has led to a variety of methods, each superior to the others in certain applications [BMSU86] [Han89a] [HanHZ89] [SaZa86] [SaZa87].

The choice of one method as the basis of our study was affected by the following three considerations.

a) facilitation of the analysis of safety of functional recursive query programs.

b) efficiency in handling functional recursive query programs.

c) flexibility in dealing with functional recursive query programs encountered in real world applications.

Based on the above considerations, we select the query-independent compilation as the basis of our research and we assume that the compilation is accomplished by the V-graph method [Han89a]. The purpose of the query-independent compilation is to reveal the regularity of the further expansions of recursions. The result of the query-independent compilation (the V-graph method) of a function-free linear recursion is a *compiled formula*. Therefore, as shown in Figure 1-3, we assume that the query-independent compilation considered in this thesis, unless otherwise specified, is accomplished by the *V-graph method*.

To facilitate the query-independent compilation which is mainly applicable to function-free recursions (e.g., function-free linear recursions), we transform functional recursive query programs into function-free ones. In order to demonstrate the generation of compiled formulas without going through the complex details of the query-independent compilation, we use the

recursive expansion technique to compile a transformed and rectified function-free linear recursion to a compiled formula. Compiled formulas are examined in Chapter 3.

From Chapter 4, our study focuses on the safety and the evaluation of compiled formulas.

It is worth noting that one of the difficulties in dealing with functional recursions is that functions are normally defined on infinite domains. Constraints have been recognized to be useful in the analysis of safety. In Chapter 4, we categorize constraints into three classes: query constraints, rule constraints and integrity constraints. Three types of integrity constraints are considered: type constraints, finiteness constraints and monotonicity constraints. We show that for functional linear recursions, safety can be viewed as a combination of two properties: finite evaluability, which guarantees the finiteness of intermediate answers, and termination, which guarantees the finiteness of the evaluation. We present a necessary and sufficient condition guaranteeing the finite evaluability of compiled formulas and a sufficient condition guaranteeing the termination of the evaluation. The algorithms for testing these conditions are developed.

In Chapter 5, we propose a safe, constraint-based evaluation method for processing compiled formulas. Based on the classification of constraints, we show that type constraints should be used to check type compatibility. Finiteness constraints can be used to generate finitely evaluable plans. Monotonicity constraints can be used in termination detection. Rule constraints should be used in compilation to reduce the search space. Query constraints are shown to be useful in the selection of efficient evaluation plans and search space reduction.

We conclude our discussion and present future research issues in Chapter 6.

A guidemap of the thesis and the basic components of our evaluation method are outlined in Figure 1-3.

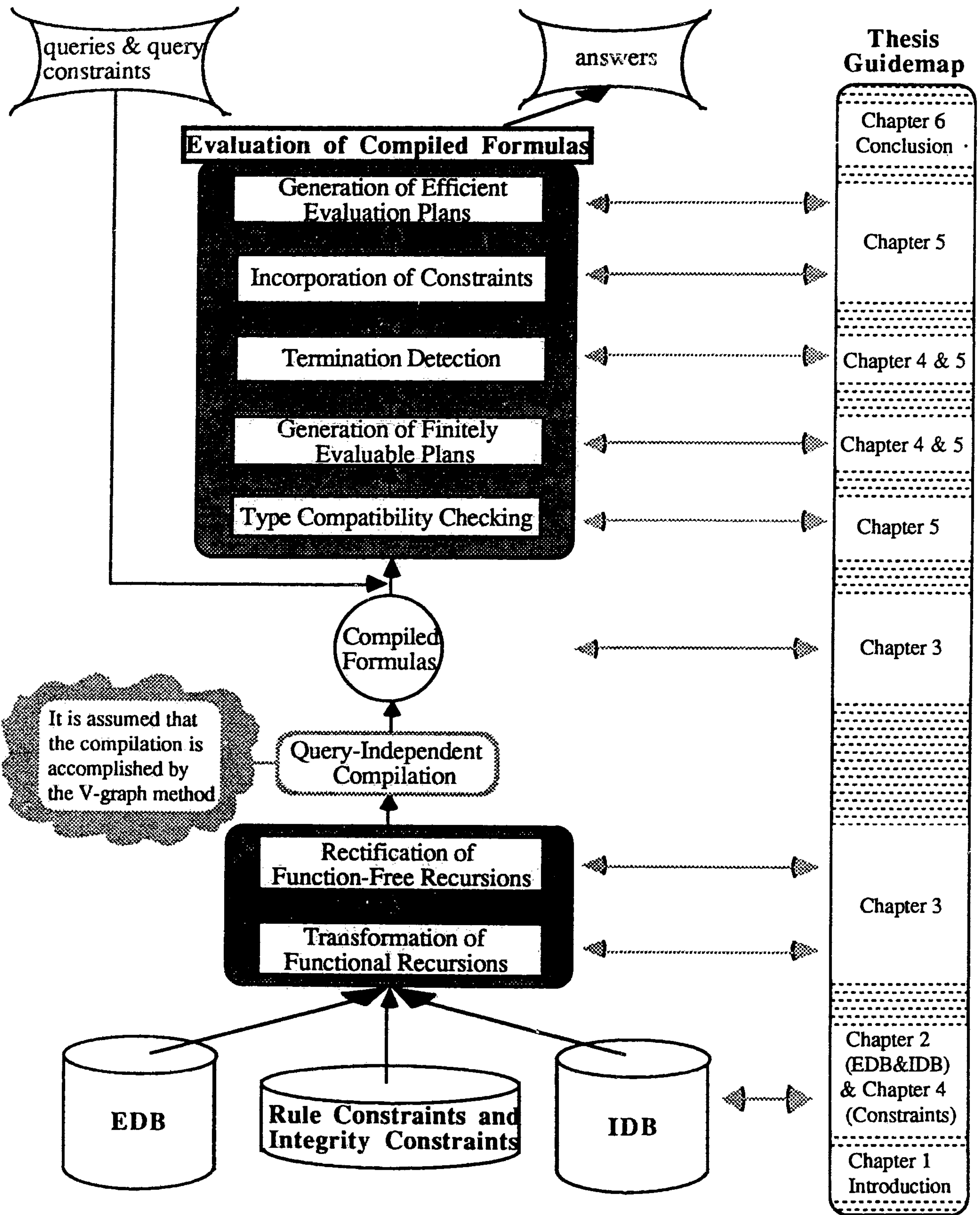


Figure 1-3 Thesis organization.

Chapter 2

Preliminaries

In this chapter we introduce Horn clause logic as a way to define deductive databases and as a language for expressing queries. As an extension of Datalog programs (i.e., function-free Horn clause programs), extended Datalog programs are introduced to deal with function symbols. Deductive databases and functions are discussed.

2.1 Deductive Databases

A deductive database is a database in which new facts may be derived from the facts that are explicitly contained in the database. In our study a deductive database is defined by a finite set of Horn clauses.

Horn clauses are clauses with at most one positive literal [PS87]. Horn clauses consist of facts, queries and rules. If a Horn clause consists of only one positive literal, it is a **fact**. If a Horn clause consists of no positive literal and one or more negative literals, it is a **query**. If a Horn clause consists of one positive literal and one or more negative literals, it is a **Horn clause rule**.

Horn Clause Rules

A Horn clause rule is of the following form:

$$r(T) \text{ :- } p_1(T_1),\ p_2(T_2),\ \ldots,\ p_n(T_n) \qquad (n \geq 0)$$

where $r(T)$ and $p_i(T_i)$ $(1 \leq i \leq n)$ are literals.

A **literal** is a predicate name with a list of arguments, each of which is a term. A **term** is a constant, a variable, or an m-ary function symbol followed by m arguments (terms). A term is **ground** if it contains no variables.

The literal $r(T)$ is called the **head** (consequent) of the rule, and the rest of the rule is called the **body** (antecedent) in which each literal $p_i(T_i)$ $(1 \leq i \leq n)$ is called a **subgoal.**

If there exists an i $(1 \leq i \leq n)$ such that $p_i = r$, the above rule is called a **recursive rule** and r is called a **recursive predicate**; otherwise, it is called a nonrecursive (**exit**) rule. If there is only one p_i $(1 \leq i \leq n)$ such that $p_i = r$, it is called a **linear recursive rule.** A **recursion**[2] of r is a finite set of Horn clause rules defining the recursive predicate r. A **recursion** is **linear** if it consists of *one* linear recursive rule and one or more exit rules [Han91].

A predicate logic is **function-free** if it does not contain function symbols. Otherwise, it is **functional** [MaW88] [BeLe86]. Similarly, we distinguish a **function-free Horn clause** from a **functional Horn clause** and a **function-free recursion** from a **functional recursion.**

Therefore, a **functional linear recursion** is a linear recursion with function symbols.

The semantics associated with the above rule is that $r(T)$ is true if all the $p_i(T_i)$'s are true (i.e., a tuple τ is in the relation for $r(T)$ if for all i $(1 \leq i \leq n)$ tuple τ_i is in the relation for $p_i(T_i)$ and we regard $r(\tau)$ as a *derived* fact); or, in the case when $n=0$, $r(T)$ is true and we regard an instance of $r(T)$ as a *base* fact.

Throughout this thesis, we use the following conventions (similar to Prolog [StSh86]) to present our examples. Predicate names, function symbols and constants start with a lower-case

---

[2] Notice that we ignore mutual/indirect recursions here [HanHY88].

letter but constants are also permitted to be numbers. A variable must start with a capital letter or is "_" [PS87]. The relation for a predicate *p* is denoted by *rel(p)*. In addition, the arithmetic comparison predicates, =, ≤, and so on, are referred to as **built-in predicates** [Ullm89b].

Example 2-1: Consider the following Horn clauses:

> *natural_number(0).*
>
> *natural_number(succ(X)) :- natural_number(X).*

where *succ(X)* is a function which returns the successor of *X*.

The meaning of the above Horn clauses is that *0* is a natural number, and *succ(X)* is a natural number if *X* is a natural number.

The first rule is an exit rule and the second one is a linear recursive rule. By definition, this program is a linear recursion with one function, *succ(X)*.

□

Datalog Programs and Extended Datalog Programs

By a **Horn clause logic program**, we mean a finite set of Horn clause rules. Due to the importance of function symbols in recursive query programs, we categorize Horn clause logic programs into two classes: **Datalog programs**, in which function symbols are not allowed, and logic programs with function symbols. To facilitate the analysis of logic programs with function symbols, we use **extended Datalog programs** (EDPs) as our study model. An *extended Datalog program* is an abstracted Datalog program[3] in which function symbols are represented by possibly infinite relations *with integrity constraints* specified for the functions (particularly, type

[3] The difference between a Datalog program and an extended Datalog program is that an extended Datalog program not only uses possibly infinite relations to represent function symbols but also associates their integrity constraints with the corresponding relations.

constraints, finiteness constraints and monotonicity constraints) [BroS89b] [KRS88] [KiRS88]. Therefore, Datalog is a subset of extended Datalog (Figure 2-1). In Chapter 3, we will examine how to transform a logic program with function symbols into an EDP.

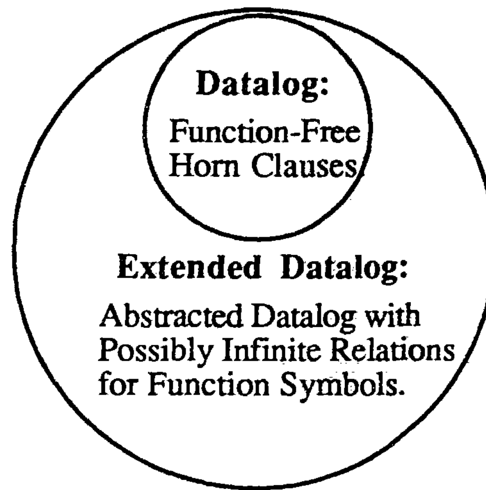


Figure 2-1 Datalog and extended Datalog.

Example 2-2: Consider the following typical Datalog program [BaRa86] [BMSU86] [Ullm89b].

same_generation(X,X) :- *person(X).*

same_generation(X,Y) :- *parent(X,X_1), same_generation(X_1,Y_1), parent(Y,Y_1).*

where *parent(X,Y)* and *person(X)* are predicates indicating that *Y* is the parent of *X* and *X* is a person respectively.

The meaning of the above program is that *X* and *Y* are in the same generation if $X = Y$ and *X* is a person (the first rule), or *X* and *Y* have parents X_1 and Y_1 respectively and X_1 and Y_1 are in the same generation (the second rule).

Since the above program consists of two Horn clause rules without any function symbol, it is a Datalog program.

□

Example 2-1 is a functional linear recursion. The following example is an extended Datalog program.

Example 2-3: Consider the following extended Datalog program.

$list(L) :- L = [].$

$list(L) :- list(L_1), cons(H,L_1,L).$

where we use predicate[4] $cons(H,L_1,L)$ to represent the list construction function $L = [H/L_1]$ which concatenates the head element H with the list L_1 to form a new list L.

The meaning of the above program is that L is a list if L is empty, or L_1 is a list and L is the resulting list of concatenating the head element H with the list L_1 .

The above program is an EDP, in which the list construction function $L = [H/L_1]$ is represented by the predicate $cons(H,L_1,L)$ whose relation is infinite and is associated with the integrity constraints specified for the function (e.g., H is a character, L_1 and L are strings of characters, etc.).

□

Queries

A **query** is a Horn clause with only negative literals and is expressed as $r_1, r_2, ..., r_n$ with $n > 0$.

There are three possible assignments to an argument of a predicate in a query: an *instantiated value* given by the query, a *variable name* indicating the expected answers to the query at that position and a *don't care symbol*, denoted by "_", indicating that the query does not care

---

[4] This predicate is called functional predicate and will be further examined in Chapter 3.

the results at that position. For explicit representation of a query, "?-" is normally placed in front of the query.

In the study of query processing, it has been realized that some queries just want to check the existence of some given facts in the database. For these kind of queries (called **existence checking queries**), the evaluation does need to continue if the existence conditions are satisfied. The techniques for recognizing existence checking queries and processing them efficiently have been studied intensively [Han89b]. Notice that if every argument in a query is either instantiated or specified by "_", this query is an existence checking query.

We examine an example.

Example 2-4: Consider the following functional linear recursion.

descendant(parent(Y),Y).

descendant(parent(X),Y) :- descendant(X,Y).

where *parent(X)* is a function which returns the parent of *X* by, for instance, looking at a pre-stored table which maintains the parent of *X*.

The meaning of the above program is that *Y* is the descendant of its parent (the first rule), *or Y* is the descendant of the parent of *X* if *Y* is the descendant of *X* (the second rule).

Consider the following queries:

query 1: *?-descendant(john,alice).*

query 2: *?-descendant(john,Y).*

query 3: *?-descendant(john,_).*

Query 1 is an existence checking query which asks if *alice* is an descendant of *john*. Query 2 asks who the descendant of *john* is. Query 3 is also an existence checking query which

asks if *john* has a descendant (but the query doesn't care who the descendant of *john* is. The evaluation does not need to continue so long as one descendant of *john* is found[5]).

Deductive Databases

Deductive databases are defined by first-order databases. A first-order database is a database consisting of first-order clauses. When a first-order database consists only of Horn clauses which are definite assertions and definite data, it is called a **definite deductive database**. When it contains indefinite assertions and indefinite data, it is called an **indefinite deductive database**. An indefinite assertion is an assertion whose consequent part consists of a disjunction of literals, and indefinite data contain facts represented by disjunctions of literals. Our research deals only with definite deductive databases and the term **deductive databases** used in this thesis refers to **definite deductive databases**.

A predicate whose relation is a set of tuples consisting only of constants stored in the database is called an *extensional database predicate* (**EDB predicate**). A predicate which is defined by Horn clause rules (i.e., it appears in the head of some rule) is called an *intentional database predicate* (**IDB predicate**). For instance, in Example 2-2, *person(X)* and $parent(X,X_1)$ are EDB predicates, but *same_generation(X,Y)* is an IDB predicate.

A **deductive database** is assumed to consist of three portions [BaRa86] [BMSU86]: (i) an *extensional database* (**EDB**) which is a set of relations for EDB predicates, (ii) an *intentional database* (**IDB**) which is a set of relations for IDB predicates, and (iii) a set of *integrity constraints* (**ICs**) specified over the EDB predicates and infinite relations for function symbols. Thus a deductive database can be represented by a triple (EDB,IDB,ICs).

---

5 The detailed study of existence checking queries can be found in [Han89b].

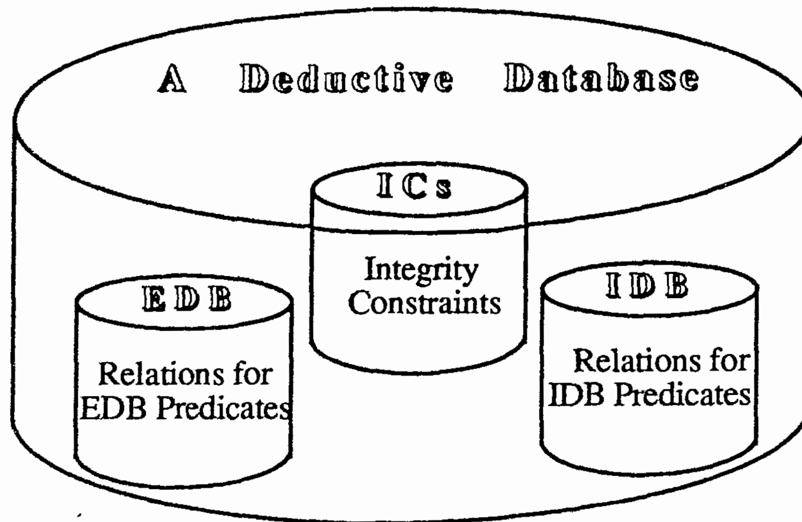


Figure 2-2 Deductive databases.

The following example describes a small deductive database.

Example 2-5: Consider the following *travel* program which describes a deductive database for Example 1-1.

$travel([Fno],Dep,DTime,Arr,ATime,Fare)$:- $flight(Fno,Dep,DTime,Arr,ATime,Fare)$.

$travel([Fno|L_1],Dep,DTime,Arr,ATime,Fare)$:- $flight(Fno,Dep,DTime,Dep_1,IATime,S)$,

$travel(L_1,Dep_1,IDTime,Arr,ATime,Fare_1)$,

$Fare = S + Fare_1$.

where $flight(Fno,Dep,DTime,Arr,ATime,Fare)$ is an EDB predicate whose relation is shown in Table 1-1. The total fare *Fare* is calculated by an arithmetic function which sums up the fares of the connecting flights, that is, $Fare = S + Fare_1$. The sequence (list) of connecting flight numbers is generated by the list construction function $[Fno|L_1]$, which takes an element *Fno* as the head and a list L_1 as the rest of the resulting list.

The meaning of the above program can be interpreted as follows:

The first rule: travel with fare *Fare* between departure city *Dep* at *DTime* and arrival city *Arr* at *ATime* can be arranged if there is a direct flight between the two cities, or

<u>The second rule:</u> such travel can be arranged if there is a flight *Fno* from city *Dep* at *DTime* to city Dep_1 with cost *S* and the travel with fare $Fare_1$ between city Dep_1 and city *Arr* can be arranged. The total fare is $Fare = S + Fare_1$ and the list of flight numbers is $[Fno/L_1]$.

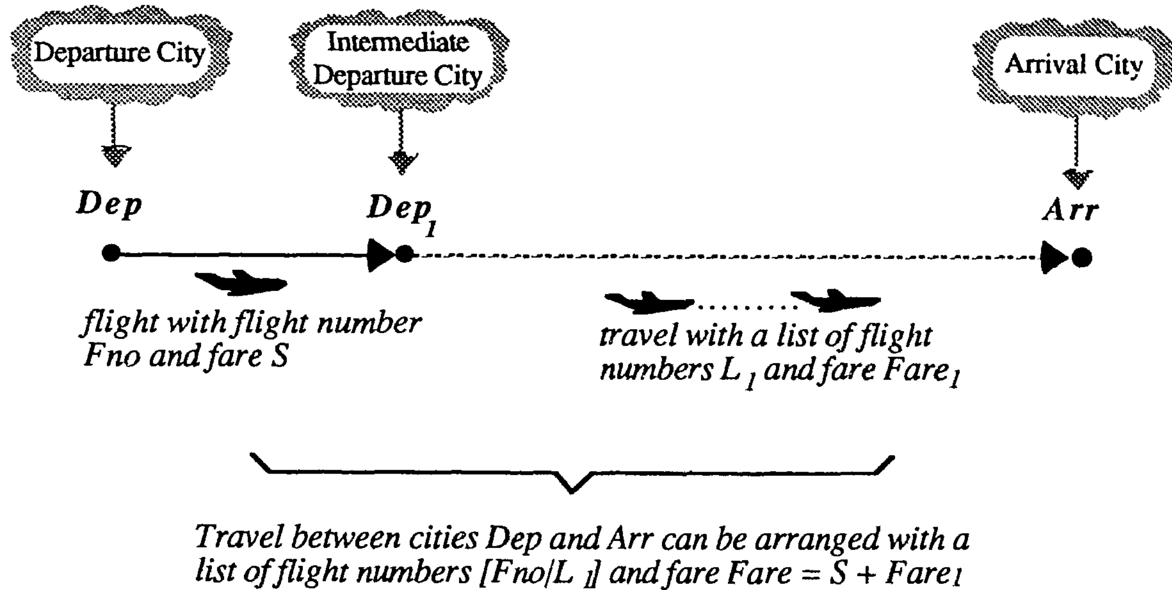


Figure 2-3 Illustration of the *travel* program.

In this small deductive database, the *extensional database* contains the relation for the EDB predicate *flight(Fno,Dep,DTime,Arr,ATime,Fare)*. The *intentional database* consists of the derived relation for the IDB predicate $travel([Fno/L_1],Dep,DTime,Arr,ATime,Fare)$[6]. $Fare = S + Fare_1$ and $[Fno/L_1]$ are two functions. The following *integrity constraints* could be specified over the EDB predicate *flight* and the infinite relations for the functions $Fare = S + Fare_1$ and $[Fno/L_1]$: (i) The types of the variables *Fare*, *S* and $Fare_1$ are positive real numbers, (ii) $[Fno/L_1]$ is known if *Fno* and L_1 are known, and (iii) the value of *Fare* is greater than the values of *S* and $Fare_1$. Detailed discussion of integrity constraints will be presented in Chapter 4.

Now we use queries to represent the questions asked by the customer in Example 1-1.

---

[6] Note that *travel([Fno],Dep,DTime,Arr,ATime,Fare)* is just a special case of $travel([Fno/L_1],Dep,DTime,Arr,ATime,Fare)$ with $L_1 = []$.

Question 1: *"Could you help me find a sequence of connecting flights from Vancouver to Ottawa?"*

query 1: *?-travel(L,vancouver,_,ottawa,_,_).*

where we use a variable *L* to indicate the expected list of flight numbers.

Question 2: *"I'd like to leave Vancouver between 7am and 8am for Ottawa. Could you arrange that for me?"*

query 2: *?-travel(L,vancouver,DTime,ottawa,_,_), 7:00 ≤ DTime, DTime ≤ 8:00.*

Question 3: *"I have an appointment in Ottawa at 2pm and I want to have lunch before the meeting. Can I get a flight, or connecting flights from Vancouver and arrive in Ottawa around noon?"*

query 3: *?-travel(L,vancouver,_,ottawa,ATime,_), 11:45 < ATime, ATime < 12:15.*

where *11:45 < ATime* and *ATime < 12:15* mean that the arrival time *ATime* is around noon.

Question 4: *"My boss can only reimburse me $500 for my travel from Vancouver to Ottawa. Can I get a ticket for under $500?"*

query 4: *?-travel(L,vancouver,_,ottawa,_,Fare), Fare ≤ 500.*

Question 5: *"I want to go from Vancouver to Ottawa. I want all intermediate stops to be at international airports. Please arrange that for me."*

Since this question requires that every intermediate city have an international airport, we need to update the original program to express such a *constraint*. This situation will be discussed in Chapter 4.

□

2.2 Functions in Deductive Databases

Data can be represented by relations and functions. The integration of relational and functional programming languages has been intensively studied [BeLe86]. It is known that functions can be represented by relations [Gru89]. We assume that the semantics of an EDP is restricted to the ground interpretation of the EDP (i.e., all the relations and the derived answer sets consist only of ground facts).

Functions

An m-ary **function** [Ullm89b] [STZ88] is a statement of the form:

$$f(T_1, T_2, \ldots, T_m). \qquad (m > 0)$$

where f is a function symbol and T_i $(1 \leq i \leq m)$ is a constant, a variable, or a function.

In deductive databases, functions are extended to allow single value or multiple values to be returned by a function [AH88] [STZ88]. For instance, in Example 2-5, the function $Fare = F_1 + F_2$ returns one value when F_1 and F_2 are given, and in Example 2-4, the function *parent(X)* can return the name of a person's father, mother or both when X is given.

The representation and implementation of functions have been intensively studied [Gru89] [AH88] [STZ88] [BeLe86] [BNRST87] [TsZa86]. When multiple values are allowed in a function, a sorted list can be used to represent this function [STZ88]. If an argument or the returned result of a function allows multiple values[7], for example $\{X_1, X_2, \ldots, X_m\}$, then it can be represented by a list $[X_1, X_2, \ldots, X_m]$. However, as we can see, such a representation is not equivalent, e.g., *{john,alice}* = *{alice,john}*, but *[john,alice]* ≠ *[alice,john]*. To deal with this

[7] However, the number of values is assumed to be finite.

situation, a sorted list can be employed to represent these values. The unification between these values and a sorted list can be performed after these values are sorted. Such a technique, called the *compile time rewriting technique*, is used by the LDL prototype [BNRST87] [STZ88].

Therefore, given a function $f(T_1,T_2,...,T_m)$, one can use a relation[8], say $f(T_1,T_2,...,T_m,V)$ where V holds/unifies the returned value(s) of the function, to represent it with the following semantics: a tuple $(\tau_1,\tau_2,...,\tau_m,v_0)$ is in the relation $f(T_1,T_2,...,T_m,V)$ iff $v_0 = f(\tau_1,\tau_2,...,\tau_m)$. Such a representation leads to the transformation of functional Horn clauses to their function-free counterparts and will be discussed further in Chapter 3.

Functions in Applications

Functions have been shown to be useful in many cases [Gru89] [AH88] [STZ88], such as arithmetic calculations, list operations and even the manipulation of complex objects [AG88]. Table 2-1 lists some commonly used functions. Many of them are supported by Prolog III [Col90]. More complicated functions can be constructed based on these functions. Special functions used in specific applications, such as a geographic information system, may not appear in this table.

Although the types of functions seem to be distinct, they still have some properties in common. For instance, for some functions, when $T_1,T_2,...,T_m$ are given, $f(T_1,T_2,...,T_m)$ can be computed, and some T_i can also be computed when other T_j's $(j \neq i)$ or $f(T_1,T_2,...,T_m)$ are known (e.g., for function $Fare = S + Fare_1$ discussed in Example 2-5, when $Fare$ and $Fare_1$ are known, S can be computed). This property is abstracted as the *finiteness constraint*. Another

---

[8] To make the explicit linkage between a function and its relational representation, we use the name of the function to represent its corresponding relation. However, a renaming technique should be used whenever an ambiguity occurs.

property shared by some functions is *monotonic behavior*. For example, for function $Fare = S + Fare_1$, given S, when the value of $Fare_1$ increases (decreases), so does the value of $Fare$.

	Function	Explanation
Numerical Operations	+	Plus
	—	Minus
	×	Times
	÷	Divide
List Operations	*[H|L]*	Concatenate the head element *H* and the list *L*
	length	Length of a list
Mathematical Functions	***round***	Round up to the closest integer
	sqr	Square
	sqrt	Positive square root
	succ	Successor

Table 2-1 Some commonly used functions.

However, not every function has these properties. For example, for function $Y = round(X)$ which rounds X up to its closest integer Y, when Y is given, there are still infinite possibilities for X, and X and Y normally do not have certain monotonic behavior.

The following example shows the two properties shared by the list construction function $L = [H/L_1]$. A more detailed discussion of these properties will be presented in Chapter 4.

Example 2-6: Consider the following function.

$$L = [H/L_1]$$

which concatenates the head element H and the list L_1 to form a new list L.

When L is given, H and L_1 are known. Given H, when the length of list L_1 increases (or decreases), so does the length of list L.

□

At the end of this chapter, we present an application example to show the use of functions in a geographic information system where a typical application is to find the area of a region.

Example 2-7: Consider the following functional linear recursion.

$area([P_1,P_2,P_3],S)$:- $triangle([P_1,P_2,P_3],S)$.

$area([P_1,P_2,P_3,Rest],A)$:- $triangle([P_1,P_2,P_3],S)$, $area([P_1,P_3,Rest],A_1)$, $A = S + A_1$.

$$triangle([P_1,P_2,P_3],S) \text{ :- } S = sqrt(V \times (V - length(P_1,P_2)) \times (V - length(P_2,P_3)) \times (V - length(P_1,P_3))).$$

where $V = \dfrac{length(P_1,P_2) + length(P_2,P_3) + length(P_1,P_3)}{2}$, $length(X,Y)$ is a function which can compute the length between two points X and Y, and $sqrt(X)$ is a function which returns the square root of X. It is assumed that the polygons are convex polygons and the point set of a polygon is represented by a sorted list.

The meaning of the above program can be interpreted as follows.

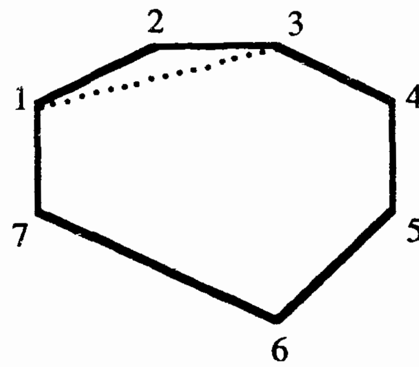
The first rule: the area of a convex polygon is that of a triangle if the polygon consists of only three points, or

The second rule: if the polygon consists of more than three points, $[P_1,P_2,P_3,Rest]$, the area of the polygon is the sum of two parts: (i) the area of the triangle with the points P_1, P_2 and P_3, and (ii) the area of a smaller convex polygon with the points P_1, P_3 and the points in the list $Rest$.

The area of a triangle is computed by the third rule.

The following query asks what is the area of the convex polygon with points *1*, *2*, *3*, *4*, *5*, *6*, and *7* (Figure 2-4).

query: *?-area([1,2,3,4,5,6,7],A).*



The area of *[1,2,3,4,5,6,7]* is the sum of the areas of *[1,2,3]* and *[1,3,4,5,6,7]*.

Figure 2-4 Illustration of computing the area of a convex polygon.

□

Chapter 3

Transformation of Functional Recursions and Compiled Formulas

In the preceding chapter, we classify Horn clause logic programs into two classes; Datalog programs and logic programs with function symbols. To facilitate the analysis of function symbols, we introduced extended Datalog in which functions are represented by possibly infinite relations (see Figure 3-1). In Section 3.1, we will discuss the transformation of logic programs with function symbols into EDPs.

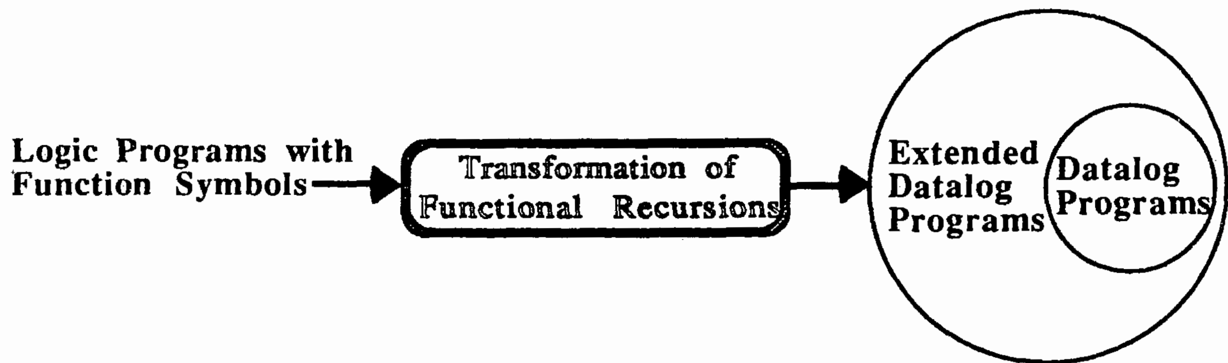


Figure 3-1 Transformation of functional recursions.

However, as suggested in [Ullm89b], logical rules in different forms should be rectified to facilitate query analysis. More importantly, the query-independent compilation takes rectified function-free recursions as input and generates compiled formulas [Han89a] [HanHY88]. Intuitively, the rules in a recursion are rectified if the heads of the rules have the same form. We will discuss Ullman's rectification technique formally in Section 3.2.

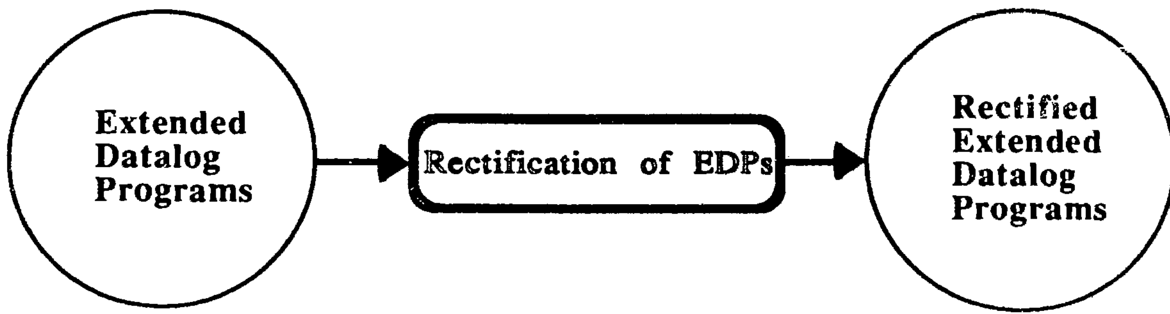


Figure 3-2 Rectification of EDPs.

Based on the rectified EDPs, we introduce the compiled formula assumption. A *compiled formula* is the result of performing the query-independent compilation (the V-graph method) on a rectified function-free linear recursion (see Figure 3-3). The purpose of the query-independent compilation is to reveal the regularity of the further expansions of recursions. However, to demonstrate the results of the query-independent compilation without going through its complex details which have been studied intensively elsewhere [Han89a], we use the **recursive rule expansion technique** to compile a rectified function-free linear recursion to a compiled formula. The *recursive rule expansion technique* is a technique which expands a recursion to reveal the regularity of its further expansions. Compiled formulas are examined in Section 3.3.

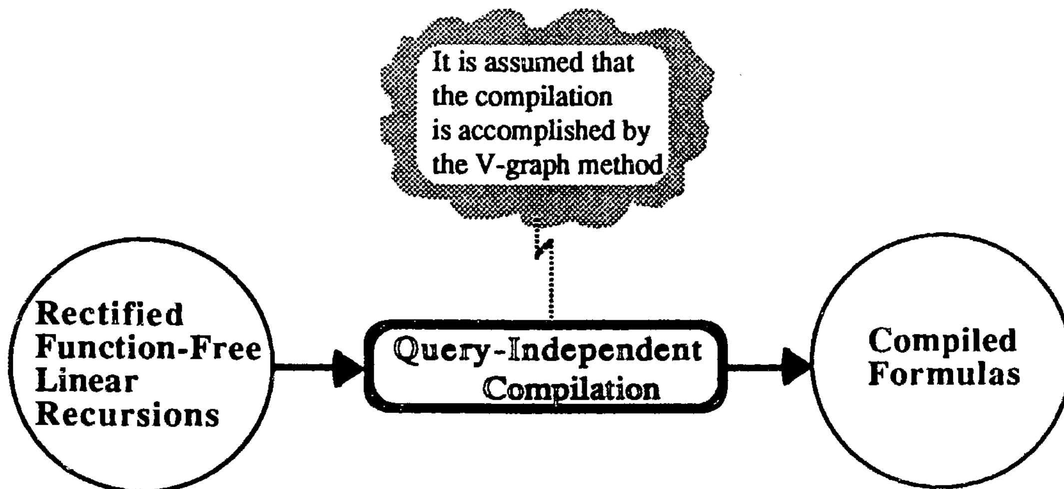


Figure 3-3 Compiled formula assumption.

3.1 Transformation of Functional Recursions

The use of relations to represent functions has been studied by several researchers [DL86] [KRS88]. In this section we discuss the transformation of functional recursions into EDPs.

The purpose of transforming functional recursions into function-free ones is to facilitate their analysis so that we can use the query-independent compilation to compile them to highly regular chain forms. The basic idea of such a transformation is to use (possibly infinite) relations to represent functions.

There are some other important reasons for transforming functional recursions into their function-free counterparts:

(a) Kifer [Kif88] and Shmueli [Shm87] have shown that for Horn clause queries with function symbols, there does not exist a general-purpose algorithm which can enumerate all answers and then terminate. Many researchers [BroS89a] [BroS89b] [RBS87] [Kif88] have shown that by transforming functional recursions to function-free ones, some query programs (e.g., monadic programs [SaVa89]) become *capturable*[9]. Our study in Chapter 4 and Chapter 5 also shows that such a transformation enables us to analyze what kind of queries are capturable.

(b) The query-independent compilation has been shown to be very useful in generating efficient evaluation plans, but is only applicable to function-free recursions [Han89a] [HanHY88] [HanH87]. By transforming functional recursions to function-free ones, we can extend the application domain of the query-independent compilation to functional recursions.

---

[9] A query is said to be capturable if there exists an algorithm which can enumerate all answers and then terminate [Kif88].

(c) Since the transformed programs are the internal representations of the original programs, the expressive power and the beauty of using functions in logic programs are preserved.

The Function-Predicate Transformation

A function-predicate transformation, called the **FP-transformation,** is a mapping from a functional recursion to a function-free one.

Algorithm 3-1: FP-transformation Algorithm.

INPUT: A rule r with function symbols.

OUTPUT: A function-free rule r'.

METHOD:

foreach function in r, say $f(X_1,X_2,...,X_m)$ **do**

begin

replace the function by a new variable, say V;

add the subgoal $f(X_1,X_2,...,X_m,V)$ in the body of the rule (called **functional predicate**[10]) with the following semantics: a tuple $(x_1,x_2,...,x_m,v_0)$ is in the relation for the functional predicate $f(X_1,X_2,...,X_m,V)$ iff $v_0 = f(x_1,x_2,...,x_m)$;

associate the integrity constraints specified for the function with the relation for $f(X_1,X_2,...,X_m,V)$

end

□

From the above algorithm we can see the FP-transformation does not change the semantics of the original programs. We examine an example below.

[10] To make an explicit linkage between a function and its functional predicate, we use the name of the function to represent its functional predicate. If there is a predicate in the original program carrying the same name but with one more argument than the function, we should rename the transformed functional predicate to avoid ambiguity.

Example 3-1: Consider the following *travel* program discussed in Example 2-5.

$$travel(Fno,Dep,DTime,Arr,ATime,Fare) :- flight(Fno,Dep,DTime,Arr,ATime,Fare).$$

$$travel([Fno|L_1],Dep,DTime,Arr,ATime,Fare) :- flight(Fno,Dep,DTime,Dep_1,IATime,S),$$
$$travel(L_1,Dep_1,IDTime,Arr,ATime,Fare_1),$$
$$Fare = S + Fare_1.$$

After performing the FP-transformation on the above program, we have the following function-free version of the *travel* program.

$$travel(Fno,Dep,DTime,Arr,ATime,Fare) :- flight(Fno,Dep,DTime,Arr,ATime,Fare).$$

$$travel(L,Dep,DTime,Arr,ATime,Fare) :- flight(Fno,Dep,DTime,Dep_1,IATime,S),$$
$$travel(L_1,Dep_1,IDTime,Arr,ATime,Fare_1),$$
$$sum(S,Fare_1,Fare),\ cons(Fno,L_1,L).$$

where $sum(S,Fare_1,Fare)$ is the functional predicate transformed from the function '+' with the following semantics: $sum(S,Fare_1,Fare)$ is true iff $Fare = S + Fare_1$ for any instance of the variables $Fare$, S, and $Fare_1$. The functional predicate $cons(Fno,L_1,L)$ is transformed from the list construction function $[Fno|L_1]$ (Example 2-5) with the following semantics: $cons(Fno,L_1,L)$ is true iff $L = [Fno|L_1]$ for any instance of the variables L, Fno, and L_1.

□

If necessary, a functional predicate can also be transformed back to its functional counterpart by going through the reverse process. This is called the *predicate-function transformation* (**PF-transformation**). Such a transformation becomes necessary if we want to restore the original programs for some reason, such as semantic query optimization [CGM90], inference of integrity constraints [BroS89a], etc.

3.2 Rectification of Function-Free Recursions

Logical rules in different forms should be rectified to facilitate query analysis [Han89a] [Ullm89b]. When an IDB predicate p is defined by a set of Horn clause rules, the relation for this predicate is obtained by computing the relations for these rules, projecting onto the variables appearing in the heads, and taking the union. However, we have trouble when some of the heads with the predicate p have constants or repeated variables, e.g., $p(a,X,X)$. Therefore, Ullman in [Ullm89b] introduced a technique to rectify logic rules.

Another important reason for rectifying function-free recursions is that the query-independent compilation takes rectified function-free recursions as input and generates compiled formulas.

Definition: The rules for predicate p are **rectified** if all the heads of the rules are identical, and of the same form $p(X_1,X_2,...,X_k)$ for distinct variables $X_1,X_2,...,X_k$.

We describe the rectification technique proposed in [Ullm89b] below. It is assumed that each variable in the head of a rule must also appear in some subgoal of the rule [Ullm89b].

Rectification of Function-Free Rule

Suppose the function-free rule to be rectified is r with head $p(Y_1,Y_2,...,Y_k)$, where the Y's may be variables or constants, with repetitions allowed.

1. Replace the head of r by $p(X_1,X_2,...,X_k)$, where X's are each distinct, new variables;
2. Add the subgoal $X_i = Y_i$ for all i.

   If Y_i is a variable, eliminate the subgoal $X_i = Y_i$ and substitute X_i for Y_i wherever it is found in the rule.[11]

[11] Note that when we make such a substitution for Y_i, we cannot later make another substitution for the same variable Y_i.

It has been shown that the above rectification process does not change the semantics of the original function-free programs [Ullm89b].

Example 3-2: The following program is the result of applying the rectification process to the transformed *travel* program shown in Example 3-1.

$$travel(L,Dep,DTime,Arr,ATime,Fare) \text{ :- } flight(L,Dep,DTime,Arr,ATime,Fare).$$

$$travel(L,Dep,DTime,Arr,ATime,Fare) \text{ :- } flight(Fno,Dep,DTime,Dep_1,IATime,S), travel(L_1,Dep_1,IDTime,Arr,ATime,Fare_1), sum(S,Fare_1,Fare), cons(Fno,L_1,L).$$

Note that the heads of the above rules are now identical.

□

We examine an example by using the FP-transformation and the rectification process.

Example 3-3: The following rule set defines a functional linear recursion in which $f(x)$ and $g(y)$ are two unary functions.

$$s(X,X) \text{ :- } person(X).$$

$$s(X,Y) \text{ :- } s(f(X),g(Y)).$$

The recursion can be considered as a functional version of the *same_generation* program (Example 2-2), where *f* and *g* can be viewed as the function *parent(X)* discussed in Example 2-4.

After performing the FP-transformation, we have the following function-free recursion:

$$s(X,X) \text{ :- } person(X).$$

$$s(X,Y) \text{ :- } f(X,X_1),\ s(X_1,Y_1),\ g(Y,Y_1).$$

where $X_1 = f(X)$ is mapped to the functional predicate $f(X, X_1)$ and $Y_1 = g(Y)$ is mapped to the functional predicate $g(Y, Y_1)$.

After applying the rectification process, we have the following rectified function-free recursion:

$$s(X,Y) \text{ :- } person(X),\ Y = X.$$
$$s(X,Y) \text{ :- } f(X,X_1),\ s(X_1,Y_1),\ g(Y,Y_1).$$

where we introduce a new variable Y for the repeated variable X in the first rule and add a subgoal $Y = X$ in the body of the first rule.

□

3.3 Compiled Formula Assumption

Several techniques, such as the α-graph method [Ioan85], the V-graph method [Han89a] [HanHY88] and the matrix method [SuHen90], have been developed to compile function-free recursions. The purpose of the query-independent compilation is to reveal the regularity of the further expansions of recursions by using mathematical tools, particularly graphs and matrices. The basic idea of these methods is to analyze the connections among variables and generate highly regular chain forms [Han89a] [YHH88]. We assume that the query-independent compilation method considered in this thesis, unless otherwise specified, refers to the *V-graph method* proposed by Han [Han89a] [HanHY88].

It is known that some function-free recursions, such as linear recursions [Han89a], can be compiled into highly regular chain forms or bounded recursions. Based on the results in the preceding section, a functional recursion, after being transformed and rectified to its function-free counterpart, can be so compiled as well. Since the focus of the thesis is not on compilation,

interested readers are referred to [Han89a] for a complete treatment of the V-graph method. However, we will use the recursive rule expansion technique to show the result of the query-independent compilation.

Compiled Formulas Generated by the Recursive Rule Expansion Technique

Definitions: In the compilation of a linear recursion with the recursive predicate *r*, the **first expansion** of *r* is the recursive rule *r* itself. The ***i*th expansion** of r $(i > 1)$ is the unification of the recursive rule with the $(i-1)$th expansion of *r*. The ***0*th expanded exit rule set** is the set of nonrecursive rules of *r*. The ***i*th expanded exit rule** set, denoted by r^i, is generated by the unification of the set of nonrecursive rules of *r* on the *i*th expansion of *r*. The **compiled formula** of *r* is the union of the set of formulas generated by all the expanded exit rules of *r*.

Example 3-4: Consider the following rectified *natural_number* program which is obtained by performing Algorithm 3-1 and the rectification process on the program shown in Example 2-1.

$natural_number(X) \text{ :- } X = 0.$

$natural_number(X) \text{ :- } natural_number(X_1), succ(X_1,X).$

where $succ(X_1,X)$ is the functional predicate for the function $succ(X_1)$ with the following semantics: $succ(X_1,X)$ is true iff $X = succ(X_1)$ for any instance of the variables X and X_1. The first rule is an exit rule and the second rule is a linear recursive rule.

The *1*st expanded exit rule is: $natural_number^1(X) \text{ :- } X_1 = 0, succ(X_1,X).$

The *2*nd expanded exit rule is: $natural_number^2(X) \text{ :- } X_2 = 0, succ(X_2,X_1), succ(X_1,X).$

After the *i*th expansion, we have the following compiled formula:

$$natural_number(X) = (X = 0) \cup (\bigcup_{i=1}^{\infty} ((X_i = 0), succ^{\,i}(X_i,X_{i-1}))).$$

where

$$succ^i(X_i,X_{i-1}) = \begin{cases} succ(X_1,X) & \text{if } i = 1, \\ succ(X_i,X_{i-1}),\ succ^{i-1}(X_{i-1},X_{i-2}) & \text{if } i > 1. \end{cases}$$

□

Example 3-5: Consider the following rectified program discussed in Example 3-3.

$s(X,Y)$:- $person(X)$, $Y = X$.

$s(X,Y)$:- $f(X,X_1)$, $s(X_1,Y_1)$, $g(Y,Y_1)$.

We compile the above program by using the recursive rule expansion technique as follows.

The *1*st expanded exit rule is:

$$s^1(X,Y) \text{ :- } f(X,X_1),\ (person(X_1), X_1 = Y_1),\ g(Y,Y_1).$$

The 2nd expanded exit rule is:

$$s^2(X,Y) \text{ :- } f(X,X_1), f(X_1,X_2),\ (person(X_2), X_2 = Y_2),\ g(Y,Y_1),\ g(Y_1,Y_2).$$

After the *i*th expansion, we have the following compiled formula:

$$s(X,Y) = (person(X),\ X = Y) \cup (\bigcup_{i=1}^{\infty} (f^i(X_{i-1},X_i),\ (person(X_i),\ X_i = Y_i),\ g^i(Y_{i-1},Y_i))).$$

where

$$f^i(X_{i-1},X_i) = \begin{cases} f(X,X_1) & \text{if } i = 1, \\ f(X_{i-1},X_i),\ f^{i-1}(X_{i-2},X_{i-1}) & \text{if } i > 1. \end{cases}$$

$$g^i(X_{i-1},X_i) = \begin{cases} g(X,X_1) & \text{if } i = 1, \\ g(X_{i-1},X_i),\ g^{i-1}(X_{i-2},X_{i-1}) & \text{if } i > 1. \end{cases}$$

□

Assumption: It is assumed that a **compiled formula**[12] is of the following form:

$$r = e \cup \bigcup_{i=1}^{\infty} (p_1^i, p_2^i, \ldots, p_n^i, e)$$

where e is the union of exit rules and p_j^i $(1 \leq j \leq n)$ is:

$$p_j^i = \begin{cases} p_j & \text{if } i = 1, \\ p_j, p_j^{i-1} & \text{if } i > 1. \end{cases}$$

where p_j^i is called a **chain** and each p_j in p_j^i is called a **chain element** of p_j^i.

The compiled formula generated by the query-independent compilation (V-graph method) from a linear recursion possesses the following two properties [Han89a] [Han91]:

Property 3-1: All the chains of a compiled formula are synchronized. That is, all the chains expand to the same length during the evaluation. To reflect the nature of such synchronization, we use the following notation to denote the ith expanded exit rule of the compiled formula.

$$r^i \text{ :- } \begin{cases} e & \text{when } i = 0, \\ p_1^i, p_2^i, \ldots, p_n^i, e & \text{when } i > 0. \end{cases}$$

Thus, r can also be expressed as follows:

$$r = r^0 \cup r^1 \cup r^2 \cup \ldots$$

[12] More specifically, the compiled formula is one generated by applying the V-graph method to a function-free linear recursion [Han89a].

Property 3-2: Suppose the kth and the $(k+1)$th chain elements of a chain p_j^i have a shared variable at positions s and t respectively, then the positions s and t are fixed for any $k \geq 1$, i.e., (s,t) is an invariant, called a **chain invariant** of the chain p_j^i (see Figure 3-4 below).

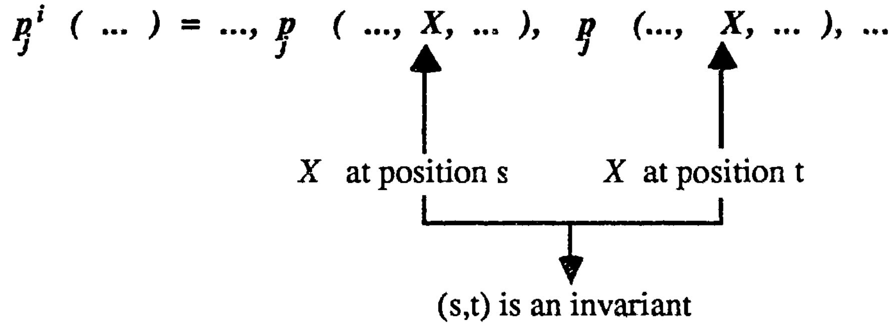


Figure 3-4 Shared variables between chain elements.

We examine an example.

Example 3-6: Consider the following rectified *append* program.

$append(U,V,W) \text{ :- } U = [], V = W.$

$append(U,V,W) \text{ :- } cons(X_1,U_1,U), append(U_1,V,W_1), cons(X_1,W_1,W).$

The *1*st expanded exit rule is:

$append^1(U,V,W) \text{ :- } cons(X_1,U_1,U), (U_1 = [],V = W_1), cons(X_1,W_1,W).$

The 2nd expanded exit rule is:

$append^2(U,V,W) \text{ :- } cons(X_1,U_1,U), cons(X_2,U_2,U_1),$
$(U_2 = [],V = W_2),$
$cons(X_1,W_1,W), cons(X_2,W_2,W_1).$

After the *i*th expansion, we have the following compiled formula:

$$append(U,V,W) = (U=[], V=W) \cup (\bigcup_{i=1}^{\infty} (cons^i(X_i,U_i,U_{i-1}), (U_i=[], V=W_i), cons^i(X_i,W_i,W_{i-1}))).$$

where

$$cons^i(X_i,U_i,U_{i-1}) = \begin{cases} cons(X_1,U_1,U) & \text{if } i = 1, \\ cons(X_i,U_i,U_{i-1}),\ cons^{i-1}(X_{i-1},U_{i-1},U_{i-2}) & \text{if } i > 1. \end{cases}$$

$$cons^i(X_i,W_i,W_{i-1}) = \begin{cases} cons(X_1,W_1,W) & \text{if } i = 1, \\ cons(X_i,W_i,W_{i-1}),\ cons^{i-1}(X_{i-1},W_{i-1},W_{i-2}) & \text{if } i > 1. \end{cases}$$

Examination of Property 3-1: the ith expanded exit rule is:

$$append^i(U,V,W) :- \begin{cases} U = [],\ V = W & \text{if } i = 0, \\ cons^i(X_i,U_i,U_{i-1}),\ (U_i = [],\ V = W_i),\ cons^i(X_i,W_i,W_{i-1}) & \text{if } i > 0. \end{cases}$$

The compiled formula can also be expressed as follows:

$append(U,V,W) = append^0(U,V,W) \cup append^1(U,V,W) \cup \ldots append^i(U,V,W) \cup \ldots$

Examination of Property 3-2: For both predicates $cons^i(X_i,U_i,U_{i-1})$ and $cons^i(X_i,W_i,W_{i-1})$, the chain invariant is (2,3) because there is one shared variable at positions 2 and 3 of two corresponding consecutive chain elements. We illustrate the chain invariant for $cons^i(X_i,U_i,U_{i-1})$ in Figure 3-5 below.

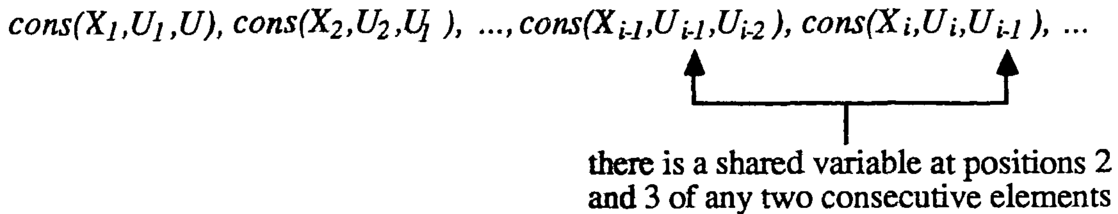


Figure 3-5 Chain invariant of $cons^i(X_i,U_i,U_{i-1})$.

□

Discussion

Compiled formulas and Properties 3-1 and 3-2 are important results of previous studies of the query-independent compilation [Han89a] [Han88b]. To capture the characteristic of a recursion, we introduce the following terms.

Definitions: A linear recursion is an ***n*-chain recursion** if for any positive integer *k*, there exists a *k*th expansion of the recursion consisting of one chain (when $n = 1$) or *n* synchronous (of the same length) chains (when $n > 1$) each with the length greater than *k*, and possibly some other predicates which do not form a chain. It is a **single-chain recursion** when $n = 1$, or **multi-chain recursion** when $n > 1$. The recursive rule of an *n*-chain recursion is called an ***n*-chain recursive rule**. A recursion is bounded if it is equivalent to a set of nonrecursive rules.

The compiled formula assumption and their properties are mainly based on the previous studies on the V-graph methods [Han89a] [YHH88] [HanHY88]. It is known that the V-graph method can compile the most commonly used recursions, linear recursions, to compiled formulas which match the representation of the compiled formulas we assumed and possess Property 3-1 and Property 3-2[13].

As shown previously, the result of performing the FP-transformation and the rectification process on a functional recursion is a rectified EDP. Consequently, we can extend the application domain of the V-graph method from function-free recursions to functional recursions.

Since the FP-transformation and the rectification process convert a functional linear recursion to a rectified function-free one and it has been proved that a rectified function-free linear recursion can be compiled to a compiled formula using the V-graph method [Han89a], we conclude that both function-free and functional linear recursions can be compiled to compiled

[13] However, it is possible for the V-graph method to expand some recursions or rename the variables of some predicates before their regularity can be revealed. The readers who are interested in the v-graph method are referred to [Han89a] [HanHY88].

formulas by the V-graph method. This conclusion is important since most recursions encountered in applications are linear recursions [Date90] [BaRa86] [Han89a] [YHH88].

As a summary of this chapter, we examine the *travel* program.

Example 3-7: Consider the following rectified *travel* program discussed in Example 3-2.

$$travel(L,Dep,DTime,Arr,ATime,Fare) \text{ :- } flight(L,Dep,DTime,Arr,ATime,Fare).$$

$$\begin{aligned} travel(L,Dep,DTime,Arr,ATime,Fare) \text{ :- } & flight(Fno,Dep,DTime,Dep_1,IATime,S), \\ & travel(L_1,Dep_1,IDTime,Arr,ATime,Fare_1), \\ & sum(S,Fare_1,Fare),\ cons(Fno,L_1,L). \end{aligned}$$

The *1*st expanded exit rule is:

$$\begin{aligned} travel^1(L,Dep,DTime,Arr,ATime,Fare) \text{ :- } & flight(Fno,Dep,DTime,Dep_1,IATime,S), \\ & flight(L_1,Dep_1,IDTime,Arr,ATime,Fare_1), \\ & sum(S,Fare_1,Fare),\ cons(Fno,L_1,L). \end{aligned}$$

The 2nd expanded exit rule is:

$$\begin{aligned} travel^2(L,Dep,DTime,Arr,ATime,Fare) \text{ :- } & flight(Fno,Dep,DTime,Dep_1,IATime,S), \\ & flight(Fno_1,Dep_1,DTime_1,Dep_2,IATime_1,S_1), \\ & flight(L_2,Dep_2,IDTime_1,Arr,ATime,Fare_2), \\ & sum(S,Fare_1,Fare),\ sum(S_1,Fare_2,Fare_1) \\ & cons(Fno,L_1,L),\ cons(Fno_1,L_2,L_1). \end{aligned}$$

After the *i*th expansion, we have the following compiled formula:

$$\begin{aligned} travel(L,Dep,DTime,Arr,ATime,Fare) = & flight(L,Dep,DTime,Arr,ATime,Fare) \cup \\ & \bigcup_{i=1}^{\infty} (flight^i(Fno_{i-1},Dep_{i-1},DTime_{i-1},Dep_i,IATime_{i-1},S_{i-1}), \\ & flight(L_i,Dep_i,IDTime_{i-1},Arr,ATime,Fare_i), \\ & sum^i(S_{i-1},Fare_i,Fare_{i-1}),\ cons^i(Fno_{i-1},L_i,L_{i-1})). \end{aligned}$$

where

$flight^i(Fno_{i-1},Dep_{i-1},DTime_{i-1},Dep_i,IATime_{i-1},S_{i-1})$

$$= \begin{cases} flight(Fno,Dep,DTime,Dep_1,IATime,S) & \text{if } i = 1, \\ flight(Fno_{i-1},Dep_{i-1},DTime_{i-1},Dep_i,IATime_{i-1},S_{i-1}), & \\ flight^{i-1}(Fno_{i-2},Dep_{i-2},DTime_{i-2},Dep_{i-1},IATime_{i-2},S_{i-2}) & \text{if } i > 1. \end{cases}$$

$$sum^i(S_{i-1},Fare_i,Fare_{i-1}) = \begin{cases} sum(S,Fare_1,Fare) & \text{if } i = 1, \\ sum(S_{i-1},Fare_i,Fare_{i-1}),\ sum^{i-1}(S_{i-2},Fare_{i-1},Fare_{i-2}) & \text{if } i > 1. \end{cases}$$

$$cons^i(Fno_{i-1},L_i,L_{i-1}) = \begin{cases} cons(Fno,L_1,L) & \text{if } i = 1, \\ cons(Fno_{i-1},L_i,L_{i-1}),\ cons^{i-1}(Fno_{i-2},L_{i-1},L_{i-2}) & \text{if } i > 1. \end{cases}$$

<u>Examination of Property 3-1:</u> the ith expanded exit rule is:

$travel^i(L,Dep,DTime,Arr,ATime,Fare)$

$$:- \begin{cases} flight(L,Dep,DTime,Arr,ATime,Fare) & \text{if } i = 0, \\ flight^i(Fno_{i-1},Dep_{i-1},DTime_{i-1},Dep_i,IATime_{i-1},S_{i-1}), & \\ flight(L_i,Dep_i,IDTime_{i-1},Arr,ATime,Fare_i), & \\ sum^i(S_{i-1},Fare_i,Fare_{i-1}),\ cons^i(Fno_{i-1},L_i,L_{i-1}). & \text{if } i > 0. \end{cases}$$

Therefore, the compiled formula can also be expressed as follows:

$$\begin{aligned} travel(L,Dep,DTime,Arr,ATime,Fare) = {} & travel^0(L,Dep,DTime,Arr,ATime,Fare) \cup \\ & travel^1(L,Dep,DTime,Arr,ATime,Fare) \cup \\ & \dots \\ & travel^i(L,Dep,DTime,Arr,ATime,Fare) \cup \\ & \dots \end{aligned}$$

Examination of Property 3-2: For the chains $sum^i(S_{i-1}, Fare_i, Fare_{i-1})$ and $cons^i(Fno_{i-1}, L_i, L_{i-1})$, the chain invariant is (2,3) because there is one shared variable at positions 2 and 3 of two corresponding consecutive chain elements. For the chain $flight^i(Fno_{i-1}, Dep_{i-1}, DTime_{i-1}, Dep_i, ATime_{i-1}, S_{i-1})$, the chain invariant is (2,4) because there is one shared variable at positions 2 and 4 of its two consecutive chain elements.

□

Chapter 4

Safety of the Evaluation of Compiled Formulas

From this chapter, our discussion focuses on compiled formulas in which the relations for functional predicates could be infinite.

It is assumed that EDB predicates and functional predicates satisfy some constraints. Three classes of constraints are considered: integrity constraints, rule constraints and query constraints. This chapter discusses the conditions under which the evaluation of a compiled formula is safe.

The evaluation of a compiled formula is **safe** with respect to a given query and a set of constraints if, for a given database instance that satisfies the constraints, the evaluation is guaranteed to yield a finite set of answers in finite steps.

Over finite databases, a bottom-up evaluation of Datalog queries is guaranteed to be safe, yielding a finite set of answers in finite steps [BaRa86] [APPRSU89] [SaVa89]. This is no longer true if some of the relations for functional predicates are infinite, since the set of answers may be infinite [BroS89b] [SaVa89].

According to Property 3-1, a compiled formula can be expressed in the following form:

$$r = \bigcup_{i=0}^{\infty} r^i$$

We show that for compiled formulas, safety can be viewed as a combination of two properties: finite evaluability, which guarantees the finiteness of intermediate answers (i.e., for

any $i \geq 0$, the relation for r^i contains a finite number of tuples during the evaluation), and termination, which guarantees the finiteness of the evaluation (i.e., there exists an m $(m \geq 0)$ such that $r = \bigcup_{i=0}^{m} r^i$).

This chapter is organized as follows. In Section 4.1 we discuss the concept of safety as used in the literature. To facilitate the analysis of safety, we discuss constraints in Section 4.2. Starting from Section 4.3, we present the major results of this chapter. Section 4.3 proves formally that safety is a combination of two properties: finite evaluability and termination. Section 4.4 presents a necessary and sufficient condition guaranteeing the finite evaluability of a compiled formula. The algorithm for testing this condition is developed. Section 4.5 presents a sufficient condition guaranteeing the termination of the evaluation of a compiled formula. The algorithm for testing the condition is also developed.

4.1 The Concept of Safety

Safety is an important issue in the analysis of functional recursions since functions are normally defined on infinite domains.

However, the term "safety" has been given different meanings in the deductive database community (see Figure 4-1). Originally, it was introduced to refer to safe queries.

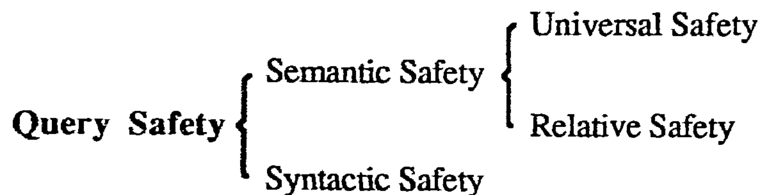


Figure 4-1 Different meanings of query safety.

The *syntactic safety* of queries was introduced in [Ullm89a] [Ullm89b] to denote a subclass of first order queries which can be translated into relational algebra, while the *semantic safety* of queries was used by some researchers to denote the class of queries with finite sets of answers [BaRa86] [KRS88] [RBS87] [Zani86]. The class of semantically safe queries properly contains the class of syntactically safe ones [Kif88]. Semantically safe queries seem to be more interesting since the finiteness of answer sets is one of the major concerns when functions are permitted.

Given a database, a query is said to be *universally safe* if the query has a finite set of answers for *every* instance of that database [Kif88]. A query is said to be *relatively safe* if the query has a finite set of answers for a *given* instance of that database [KRS88] [RBS87]. Relative safety seems to be more important since queries are always evaluated with respect to a given database instance and the user usually wants to know whether this particular evaluation generates a finite set of answers.

In order to find the answers to a query, we must use some evaluation algorithm(s). Therefore another notion, called *capturability*, was introduced [KiL88] [Ullm89b]. A query is said to be *capturable* if there exists an algorithm which can enumerate all answers and then terminate.

From a practical standpoint, "a database" normally refers to a given instance of that database. From the user's point of view, when a query is posed, one of the major considerations is whether or not there exists an evaluation algorithm which can yield a finite set of answers in finite steps. From the application point of view, constraints seem to be particularly important in

the analysis of safety since some constraints may change an infinite domain to a finite one[14]. Therefore, safety according to our definition is related to a given query and a set of constraints for a given database instance that satisfies the constraints.

From here on, the term "safety" used in this thesis refers to the safety according to our definition unless otherwise specified.

4.2 Classification of Constraints

Constraint programming and constraint-based reasoning have been studied extensively in logic programming and artificial intelligence [Coh90] [KKR90] [Las90]. It is known that constraints are also very useful in the analysis of safety in deductive databases [BroS89a] [BroS89b] [KRS88] [RBS87] [SaVa89].

We consider *constraints* as assertions that a given database instance is compelled to obey. We classify constraints (Cs) into three classes: *query constraints* (**QCs**), *rule constraints* (**RCs**) and *integrity constraints* (**ICs**). In other words, a given set of constraints is considered as **Cs = QCs ∪ RCs ∪ ICs.**

Query Constraints

A **query constraint** is a constraint which adds one or more conjuncts to the query predicates.

To facilitate the analysis of safety and query processing, we use query constraints to express the instantiated information provided by a query. More specifically, given a query:

$$?\text{-}r(\ldots,c_1,\ldots,c_2,\ldots c_i,\ldots)$$

---

14 For instance, the relation for the functional predicate *sum(S,*$Fare_1$*,Fare)* is infinite but the relation becomes finite if the following constraints are given: *S*, $Fare_1$ and *Fare* are integers, $0 \le S \le 1000$, $0 \le Fare_1 \le 1000$, and $0 \le Fare \le 1000$.

where c's are *all* the instantiated information provided by the query and i is less than or equal to the number of arguments of r, we can express this query as follows:

$$?\text{-}r(\ldots,V_1,\ldots,V_2,\ldots V_i,\ldots),\ \ V_1 = c_1,\ldots,\ \ V_2 = c_2,\ldots,\ \ V_i = c_i,\ldots$$

where each V_i is the corresponding variable name and $V_i = c_i$ is viewed as a query constraint. $?\text{-}r(\ldots,V_1,\ldots,V_2,\ldots V_i,\ldots)$ is called a **query predicate**. By using this method to express a query, we can analyze queries more precisely and select the instantiated information we need in the analysis of safety and query processing. From here on, we use this method to express queries. We examine an example below.

Example 4-1: The queries discussed in Example 2-5 can be expressed as follows.

query 1: *?-travel(L,Dep,_,Arr,_,_), Dep = vancouver, Arr = ottawa.*

query 2: *?-travel(L,Dep,DTime,Arr,_,_),*

Dep = vancouver, Arr = ottawa, 7:00 ≤ DTime, DTime ≤ 8:00.

query 3: *?-travel(L,Dep,_,Arr,ATime,_),*

Dep = vancouver, Arr = ottawa, 11:45 < ATime, ATime < 12:15.

query 4: *?-travel(L,Dep,_,Arr,_,Fare), Dep = vancouver, Arr = ottawa, Fare ≤ 500.*

where *Dep = vancouver, Arr = ottawa, 7:00 ≤ DTime, DTime ≤ 8:00, 11:45 < ATime, ATime < 12:15* and *Fare ≤ 500* are query constraints of their respective queries.

□

Rule Constraints

A **rule constraint** is a constraint which adds one or more conjuncts to the body of a deduction rule.

Example 4-2: Consider the following program [Col90].

$ElementOf(E,[E/L])$.

$ElementOf(E,[E'/L])$:- $ElementOf(E,L)$, $E \neq E'$.

where $[E/L]$ is the list construction function.

The meaning of the above program is that E is an element of $[E/L]$ if E is the head element of $[E/L]$, or E is an element of $[E'/L]$ if E is an element of L and $E \neq E'$.

$E \neq E'$ is viewed as a rule constraint since it specifies the condition that variables E and E' must satisfy.

□

Since a rule constraint adds new conjuncts (conditions) to the body of a deduction rule, it changes the original rule to a new, constrained one. Therefore, rule constraints should be compiled together with the new deduction rule. We will examine how to push rule constraints into compilation in Chapter 5.

Integrity Constraints

In this thesis, three types of integrity constraints are considered: *type constraints*, *finiteness constraints*, and *monotonicity constraints*.

Type Constraints

A **type constraint** is a constraint which requires the arguments of predicates to belong to specified domains. The type constraints for EDB predicates and the functions supported by a database are defined when the database is defined. The type constraints for IDB predicates can be

inferred when an IDB predicate is defined. We will discuss the inference of type constraints in Chapter 5. Now we examine an example.

Example 4-3: Consider the following functional recursion which finds the length of a list.

length([],0).

length([H|R],succ(N)) :- length(R,N).

The meaning of the above program is that: (i) the length of a list is *0* if the list is empty (the first rule), or (ii) the length of the list *[H|R]* is *succ(N)* if the length of *R* is *N* (the second rule).

The type constraints for the above program are: *H* is a character, *R* is a string of characters, and *N* is an integer.

□

Type constraints and the inference of type constraints

Finiteness Constraints

A **finiteness constraint** (fc) is a statement of the form

$$p: \Lambda \rightarrow \Gamma$$

where Λ and Γ are sets of the argument positions of the predicate *p* [SaVa89]. Let *rel(p)* denote the relation for the predicate *p*. Then *rel(p)* satisfies this constraint if for every tuple *t* of *rel(p)*, the set of tuples $\{s[\Gamma] | s[\Lambda] = t[\Lambda]\}$ is finite[15]. In particular, *rel(p)* satisfies $p: 0 \rightarrow \Gamma$ if $\{t[\Gamma]\}$ is finite[16]. The intuitive meaning of *rel(p)* satisfying $p: \Lambda \rightarrow \Gamma$ is that if $\{t[\Lambda]\}$ is finite, then $\{t[\Gamma]\}$

[15] Note that if $\{s[\Gamma] | s[\Lambda] = t[\Gamma]\}$ is finite, it implies that for any variable *V* whose position in *p* appears in Γ, $\{s[V]\}$ is also finite.

[16] We use *'0'* to indicate that the finiteness of $\{t[\Gamma]\}$ does not depend on any argument of *p*. For example, given an EDB predicate *parent(X,Y)*, its relation satisfies *parent:* $0 \rightarrow 1$ and *parent:* $0 \rightarrow 2$ because $\{t[1]\}$ and $\{t[2]\}$ are finite for every tuple *t* of the relation.

is finite. Clearly, for any m-ary EDB predicate p, $rel(p)$ satisfies $fc = \{p: 0 \rightarrow 1, p: 0 \rightarrow 2, \ldots, p: 0 \rightarrow m\}$ since the relation for any EDB predicate is finite and every attribute of $rel(p)$ is finite. A variable is said to be **finite** if it is bound to a finite number of facts.

Example 4-4: Consider the following functional predicate from Example 3-6.

$cons(X_1, U_1, U)$.

For any finite X_1 and U_1, there is a finite U and for any finite U there are only finite choices for X_1 and U_1. Therefore, we have the following finiteness constraints:

$$fc = \{cons: \{1,2\} \rightarrow 3,\ cons: 3 \rightarrow \{1,2\}\}.$$

Consider another functional predicate from Example 3-1.

$sum(S, Fare_1, Fare)$.

where S, $Fare_1$ and $Fare$ are positive real numbers.

Since the finiteness of any two variables makes the third variable finite (e.g., for any finite S and $Fare_1$, there is a finite $Fare$ (actually only one value for $Fare$)), we have the following finiteness constraint:

$$fc = \{sum: \{1,2\} \rightarrow 3,\ sum: \{1,3\}\} \rightarrow 2,\ sum: \{2,3\}\} \rightarrow 1\}.$$

□

Monotonicity Constraints

A **monotonicity constraint** (mc) is a statement of the form

$$p: i <_{mc} j.$$

where p is a predicate name and i and j, defined on the same domain, are argument positions of p [BroS89b]. Intuitively, the above *mc* means that the relation *rel(p)* satisfies this constraint if for every tuple t of *rel(p)*, its value at position i is less than that of j, according to some partial order. The 'less than' condition might be satisfied after performing some mapping on i or j. Interestingly, after a proper mapping is performed, relations without cyclic data (e.g., the relation for $parent(X,X_1)$ discussed in Example 2-2) always satisfy some form of mc's provided the arguments involved are defined on the same domain.

Example 4-5: Consider the following functional predicate.

$$cons(X_1,U_1,U).$$

According to the meaning of $cons(X_1,U_1,U)$ discussed in Example 3-6, the relation for $cons(X_1,U_1,U)$ satisfies the following monotonicity constraints.

$$mc = \{cons\text{: } 1 <_{mc} 3, 2 <_{mc} 3\}.$$

where $<_{mc}$ is defined on the length of lists with respect to the usual order on natural numbers.

Consider another functional predicate:

$$sum(S,Fare_1,Fare)$$

where S, $Fare_1$ and $Fare$ are positive real numbers. The relation for $sum(S,Fare_1,Fare)$ satisfies the following monotonicity constraints:

$$mc = \{sum\text{: } 1 <_{mc} 3, 2 <_{mc} 3\}$$

where $<_{mc}$ refers to the usual order on positive real numbers.

□

We use the *travel* database as an example to analyze different constraints specified over the database.

Example 4-6: Consider the following modified version of the *travel* program (Example 3-1).

$travel(L,Dep,DTime,Arr,ATime,Fare)$:- $flight(L,Dep,DTime,Arr,ATime,Fare)$.

$travel(L,Dep,DTime,Arr,ATime,Fare)$:- $flight(Fno,Dep,DTime,Dep_1,IATime,S)$,
$travel(L_1,Dep_1,IDTime,Arr,ATime,Fare_1)$,
$sum(S,Fare_1,Fare)$, $cons(Fno,L_1,L)$,
$international(Dep_1)$.

query: ?-$travel(L,Dep,_,Arr,_,_)$, $Dep = vancouver$, $Arr = ottawa$, $Fare \leq 500$.

where we assume that all the international airports are registered in the relation for $international(Dep_1)$.

Now we analyze the constraints specified over the small database.

Query Constraints:

$Dep = vancouver$, $Arr = ottawa$ and $Fare \leq 500$ are the query constraints which require that the total fare from *Vancouver* to *Ottawa* be less than or equal to *500* dollars.

Rule Constraints:

$international(Dep_1)$ is a rule constraint[17] which requires that all the intermediate airports must be international airports. Recall that question 5 in Example 1-1 needs this constraint and we mentioned that this question could not be expressed by a query (Example 2-5). This question can be described by the following query after the rule constraint is added to the *travel* program.

[17] Notice that this constraint cannot be expressed by a query constraint because the variable Dep_1 does not appear in the head of any rule in the *travel* program.

query: *?-travel(L,Dep,_,Arr,_,_), Dep = vancouver, Arr = ottawa.*

Integrity Constraints: we discuss the three types of ICs below.

Type constraints:

Variable	Type Constraints
Fno	natural number
Dep	string of char
DTime	0000..2400
Arr	string of char
ATime	0000..2400
Fare	positive real number

Table 4-1 Type constraints for the EDB predicate *flight*.

The type constraints for other predicates of the program can be inferred from the type constraints shown in Table 4-1.

Finiteness constraints:

Since *flight* is an EDB predicate, its relation is finite. We have:

$$fc_1 = \{flight\colon 0 \to 1, flight\colon 0 \to 2, flight\colon 0 \to 3, flight\colon 0 \to 4, flight\colon 0 \to 5, flight\colon 0 \to 6\}.$$

According to Example 4-4, the *fc's* for *cons* and *sum* are:

$$fc_2 = \{cons\colon \{1,2\} \to 3,\ cons\colon 3 \to \{1,2\}\}.$$

$$fc_3 = \{sum\colon \{1,2\} \to 3,\ sum\colon \{1,3\}\} \to 2,\ sum\colon \{2,3\}\} \to 1\}.$$

Thus the finiteness constraints can be expressed as follows:

$fc = fc_1 \cup fc_2 \cup fc_3.$

Monotonicity constraints:

According to Example 4-5, we have the following monotonicity constraints.

$mc_1 = \{cons: 1 <_{mc} 3, 2 <_{mc} 3\}$

where $<_{mc}$ is defined on the length of lists.

$mc_2 = \{sum: 1 <_{mc} 3, 2 <_{mc} 3\}$

where $<_{mc}$ refers to the usual order on positive real numbers.

Thus the monotonicity constraints can be expressed as follows:

$mc = mc_1 \cup mc_2.$

□

The inference of monotonicity constraints in programs is an important issue in deductive databases [BroS89a] [BroS89b]. For example, if *rel(p)* satisfies $p: i <_{mc} j$ and $p: j <_{mc} k$, then we can infer that *rel(p)* also satisfies $p: i <_{mc} k$. A sound and complete algorithm for the inference of mc's is given in [BroS89a]. Interested readers are referred to [BroS89a] and [BroS89b] for a detailed description of the algorithm.

Another issue is the use of equality constraints in the analysis of safety of logic programs [BroS89a] [BroS89b] [SaVa89]. Intuitively, the relation *rel(p)* for the predicate *p* satisfies the *equality constraint* $p: i =_{ec} j$ iff the value at position *i* and the value at position *j* are equal in every tuple of *rel(p)*. We do not discuss equality constraints in detail due to the following observation. If two variables have the same name in a Horn clause rule, then they must be defined on the same domain and have the same values in corresponding tuples. Therefore, by using the same variable name for those variables that satisfy equality constraints, equality constraints can be implicitly

expressed in a program. Moreover, the equality constraints satisfied by the variables in different predicates can also be expressed by the same variable name.

The third issue concerns the representation of rule constraints and query constraints . We assume that a rule constraint or a query constraint is given in one of following forms: $V > c$, $V \geq c$, $V = c$, $V \neq c$, $V \leq c$, $V < c$ (where V is a variable and c is a constant) with the following exceptions: (i) since an attribute V in an EDB relation contains a finite number of constants and can be expressed as $V = c$ where c is an instance of V in the relation, we also allow a rule constraint or a query constraint to be specified by an EDB relation. For example, the rule constraint $international(Dep_1)$ discussed in Example 4-6 is specified by an EDB relation which registers the names of all the international airports. (ii) Since a rule constraint will be compiled together with the original rule, we also allow it to be a function[18].

4.3 Safety as a Combination of Finite Evaluability and Termination

We view safety as a combination of two properties: finite evaluability and termination. The evaluation of a compiled formula is **safe** with respect to a given query and a set of constraints if, for a given database instance that satisfies the constraints, the evaluation is guaranteed to yield a finite set of answers in finite steps.

A predicate p is **finitely evaluable** with respect to a given query and a set of constraints iff the relation for p is finite during the evaluation, where the relation for p is from a given database instance that satisfies the constraints.

[18] Note that this function will be transformed into a functional predicate after the FP-transformation is performed on the modified, more constrained rule.

According to Property 3-1, a compiled formula r can be expressed as $r = r^0 \cup r^1 \cup r^2 \cup \ldots = \bigcup_{i=0}^{\infty} r^i$. Therefore, r is finitely evaluable with respect to a given query and a set of constraints iff the relation for r^i is finite for every i $(i \geq 0)$ during the evaluation.

The evaluation of a Horn clause program **terminates** with respect to a given query and a set of constraints if the evaluation can generate the final set of answers in finite steps, where the relations for the predicates in the program are from a given database instance that satisfies the constraints.

For a compiled formula

$$r = r^0 \cup r^1 \cup r^2 \cup \ldots = \bigcup_{i=0}^{\infty} r^i,$$

the evaluation of r terminates with respect to a given query and a set of constraints iff there exists an m $(m \geq 0)$ such that $r = \bigcup_{i=0}^{m} r^i$.

As we can see from the above definitions, for compiled formulas, safety can be viewed as a combination of finite evaluability and termination. We will examine them separately in the following two sections (i.e., Sections 4.4 and 4.5).

4.4 Safety I: Finite Evaluability

In this section, we present a necessary and sufficient condition which guarantees the finite evaluability of a compiled formula with respect to a given query and a set of constraints. The algorithm for testing the condition is developed. Before we discuss the details formally, we give the finiteness propagation rules below where the adornment b indicates that the corresponding variable is bound.

Finiteness Propagation Rules: Suppose r is a rule. Given a query and a set of constraints, we adorn r as follows.

1. If a variable V appears in an EDB predicate or $V = c$ where c is a constant or $c \in C$ where C is a finite domain, V is adorned with b, that is, V^b.
2. Two variables specified by "=" or with the same variable name are adorned with b if one of them is adorned with b.
3. If there is a finiteness constraint $p: \Lambda \rightarrow \Gamma$ and all the variables specified[19] in Λ are adorned with b, then every variable V specified in Γ is adorned with b, that is, V^b.

We examine an example below.

Example 4-7: Consider the following program which defines the arithmetic function $Z = X + Y$, where Z, X and Y are integers.

$plus(X,Y,Z)$:- $X = 0, Z = Y$.

$plus(X,Y,Z)$:- $plus(X_1,Y,Z_1)$, $succ(X_1,X)$, $succ(Z_1,Z)$.

where the finiteness constraints satisfied by *succ* are: $fc = \{succ: 1 \rightarrow 2, succ: 2 \rightarrow 1\}$

Suppose the following query is posed:

query 1: *?-plus(X,Y,Z), Y=3.*

After applying the finiteness propagation rules to the exit rule and the first expanded exit rule, we have the following adorned rules.

$plus(X^b,Y^b,Z^b)$:- $X^b = 0, Z^b = Y^b$.

$plus(X^b,Y^b,Z^b)$:-$(X_1^b = 0, Y^b = Z_1^b)$, $succ(X_1^b,X^b)$, $succ(Z_1^b,Z^b)$.

---

[19] By "a variable is specified in Λ (or Γ)", we mean that the position of that variable appears in Λ (or Γ).

Notice that after the finiteness propagation, the relation for every predicate in the above rules is finite since every variable in the above rules are bound to a finite number of facts (indicated by the adornment *b*).

Consider another query:

query 2: *?-plus(X,Y,Z), X=3.*

After applying the finiteness propagation rules to the exit rule and the first expanded exit rule, we have the following adorned rules.

$plus(X^b,Y,Z)$:- $X^b = 0, Z = Y.$
$plus(X^b,Y,Z)$:-$(X_1^b = 0, Y = Z_1), succ(X_1^b,X^b), succ(Z_1,Z).$

Notice that the variables without the adornment *b* mean that they cannot be bound to a finite number of facts. In other words, the relation for the predicate that has a variable without the adornment *b* is finite. For example, the relations for $plus(X^b,Y,Z)$ and $succ(Z_1,Z)$ are infinite with respect to query 2.

□

4.4.1 Testing of Finite Evaluability

In this Section, we present a necessary and sufficient condition which guarantees the finite evaluability of a compiled formula. The algorithm for testing the condition is also presented.

Lemma 4-1: A rule *r* is finitely evaluable with respect to a given query and a set of constraints iff for a given database instance that satisfies the constraints, every variable in the predicates of *r* is adorned with *b* after the finiteness propagation rules are applied to *r*.

Proof: The proof is straightforward in the sense that the relation for a predicate is finite iff every attribute of the relation is finite. According to the finiteness propagation rules, an attribute is bound to a finite number of facts with respect to the query and the constraints iff the attribute is adorned with b. Hence, the lemma holds.

□

Lemma 4-2: Suppose r is a compiled formula.

If a) the exit rule set is finitely evaluable, and

b) the first expanded exit rule set is finitely evaluable,

then its further expanded exit rule set is also finitely evaluable.

Proof: Suppose the compiled formula is r. According to Properties 3-1, r can be expressed as follows.

$$r = \bigcup_{i=0}^{\infty} r^i$$

where

$$r^i :- \begin{cases} e & \text{when } i = 0 \\ p_1^i, p_2^i, \ldots, p_n^i, e & \text{when } i > 0 \end{cases}$$

Therefore, according to the definition of finite evaluability, we only need to prove that for any i $(i \geq 0)$, the relation for r^i is finite.

The proof is by induction on the expansion length of the expanded exit rules of r.

For the basis, according to the condition of this lemma, the relations for r^0 and r^1 are finite.

For the induction, if the relation for r^{i-1} is finite, we prove that the relation for r^i is also finite.

According to the above equations, we can infer $r^i = p_1, p_2, ..., p_n, r^{i-1}$. Let s:-$p_1, p_2, ..., p_n$ and the relation for s be $rel(s)$. Then the relation for $rel(r^i)$ can be expressed as follows:

$$rel(r^i) = \pi(rel(s) \underset{\$i_1=\$j_1 \wedge \$i_2=\$j_2 \wedge ... \$i_m=\$j_m}{\bowtie} rel(r^{i-1}))$$

where $\$i_h$ and $\$j_h$ $(1 \leq h \leq m$, m is less than or equal to the smallest number of the attributes of $rel(s)$ and $rel(r^{i-1}))$ are the attribute positions of $rel(s)$ and $rel(r^{i-1})$ respectively. $\$i_h = \$j_h$ means that the attributes at the corresponding positions in $rel(s)$ and $rel(r^{i-1})$ have the same attribute name. π denotes the projection on the attributes that appear in $rel(r^i)$.

By the inductive hypothesis, $rel(r^{i-1})$ is finite. Hence, according to the definitions of the natural join and projection operations [Ullm89b], $rel(r^i)$ is finite.

□

Theorem 4-2: A compiled formula is finitely evaluable with respect to a given query and a set of constraints iff all the variables in the exit rule set and the first expanded exit rule set are adorned with b's after the finiteness propagation rules are applied to them.

Proof: Suppose the compiled formula is r.

Only If: According to the definition of finite evaluability, a compiled formula r is finitely evaluable with respect to a given query and a set of constraints iff $rel(r^i)$ is finite for any i $(i \geq 0)$ during the evaluation. Therefore, at least $rel(r^0)$ and $rel(r^1)$ must be finite. According to Lemma 4-1, every variable in the exit rule r^0 and the first expanded exit rule r^1 is adorned with b's after the finiteness propagation rules are applied.

If: Suppose all the variables in the exit rule set and the first expanded exit rule set are adorned with *b's* after the finiteness propagation rules are applied to them. According to Lemma 4-1, r^0 and r^1 are finitely evaluable, then according to Lemma 4-2, r^i $(i \geq 0)$ is also finitely evaluable. By definition, *r* is finitely evaluable.

□

Based on Theorem 4-2, we can develop an algorithm to test the finite evaluability of a compiled formula with respect to a given query and a set of constraints.

Algorithm 4-1: Finite Evaluability Testing Algorithm.

INPUT: A compiled formula, a query and a set of constraints.

OUTPUT: An assertion of whether the compiled formula is finitely evaluable ('yes') or not ('no').

METHOD:

```
apply the finiteness propagation rules to the exit rule set;
if every variable in the exit rule set is adorned with b
  /* in other words, every variable is bound to a finite number of facts (indicated by the
  adornment b) */
    then
      begin
        apply the finiteness propagation rules to the first expanded exit rule set;
        if every variable in the first expanded exit rule set is adorned with b
            then return('yes')
            else return('no')
      end
    else return('no')
```

□

4.4.2 Analysis of the Finite Evaluability Testing Algorithm

The correctness of Algorithm 4-1 is clear since it does nothing but test the condition of Theorem 4-2 which has been proved to be necessary and sufficient for guaranteeing the finite evaluability of a compiled formula with respect to a given query and a set of constraints. Therefore, we have the following proposition.

Proposition 4-1: The finite evaluability testing algorithm is correct in the sense that, given a compiled formula, a query and a set of constraints, Algorithm 4-1 returns 'yes' if and only if the compiled formula is finitely evaluable with respect to the query and the constraints.

□

We examine an example below.

Example 4-8: Consider the *plus* program discussed in Example 4-7.

$plus(X,Y,Z) \text{ :- } X = 0,\ Z = Y.$

$plus(X,Y,Z) \text{ :- } plus(X_1,Y,Z_1),\ succ(X_1,X),\ succ(Z_1,Z).$

$fc = \{succ\colon 1 \rightarrow 2,\ succ\colon 2 \rightarrow 1\}.$

Suppose the following query is posed:

query 1: $?\text{-}plus(X,Y,Z),\ Y=3.$

After applying the finiteness propagation rules to the exit rule and the first expanded rule, Algorithm 4-1 returns 'yes', indicating that the above program is finitely evaluable (see Figure 4-2).

$plus^0(X^b, Y^b, Z^b)$:- $X^b = 0$, $Z^b = Y^b$.

instantiated by the query ?-*plus(X,3,Z)*

(a) Finiteness propagation in the exit rule.

$plus^1(X^b, Y^b, Z^b)$:- $(X_1^b = 0, Z_1^b = Y^b)$, $succ(X_1^b, X^b)$, $succ(Z_1^b, Z^b)$.

instantiated by the query ?-*plus(X,Y,Z), Y=3.*

(b) Finiteness propagation in the first expanded rule.

Figure 4-2 Application of Algorithm 4-1 in the *plus* program.

For query 2: ?-*plus(X,Y,Z), X=3,* as shown in Example 4-7, there are variables which are not adorned with *b*. Therefore, Algorithm 4-1 returns 'no', indicating that the compiled formula is not finitely evaluable with respect to query 2.

□

4.5 Safety II: Termination

Many researchers' studies show that the termination problem is difficult to solve in some circumstances [BroS89b] [Kif88] [RBS87] [SaVa89]. Kifer [Kif88] showed that Horn clause queries with function symbols are not capturable. In other words, there does not exist an evaluation algorithm which can enumerate all answers in finite steps for these queries. The following example shows that the evaluation of a functional recursion cannot even predictably terminate.

Example 4-9: Consider the following program and query.

$r(X)$:- $a(X)$.

$r(X)$:- $random(X,X_1)$, $r(X_1)$.

query: ?-$r(X)$, $X=1$.

where $a(X)$ is an EDB predicate containing only one fact $a(0)$ and $random(X,X_1)$ is a functional predicate which generates a random number when X is given.

Since we do not know what will be generated by the function *random*, the evaluation of the above query program cannot terminate.

□

Our study shows that based on the compiled formulas generated from functional recursions by using the V-graph method, we can analyze what kinds of queries are capturable. More specifically, we show that whether or not there exists a safe execution of a compiled formula with respect to a given query and a set of constraints depends on the evaluation plan generated.

Definition: Given a compiled formula r, a query and a set of constraints, an **evaluation plan** for r is an arrangement indicating how to evaluate the chains and the exit rule set of r. To represent the evaluation order among the chains and the exit rule set of a compiled formula, we use $\Rightarrow$ to indicate that the predicate to the left of $\Rightarrow$ should be evaluated first.

We examine an example below.

Example 4-10: Consider the following program which defines the arithmetic function $Z = X \times Y$.

$times(0,Y,0)$.

$times(succ(X),Y,Z)$:- $times(X,Y,Z_1)$, $plus(Z_1,Y,Z)$.

where X, Y and Z are positive integers, *succ* is the successor function and *plus* is defined in Example 4-7.

After performing the FP-transformation and the rectification process, we have:

$$times(X,Y,Z) \text{ :- } X = 0, Z = 0.$$

$$times(X,Y,Z) \text{ :- } succ(X_1,X),\ times(X_1,Y,Z_1),\ plus(Z_1,Y,Z).$$

After the *i*th expansion, we have the following compiled formula:

$$times(X,Y,Z) = (X = 0, Z = 0) \cup \bigcup_{i=1}^{\infty} (succ^i(X_i,X_{i-1}),\ (X_i = 0, Z_i = 0),\ plus^i(Z_i,Y,Z_{i-1}))$$

where $succ^i(X_i,X_{i-1})$ is defined in Example 3-4 and

$$plus^i(Z_i,Y,Z_{i-1}) = \begin{cases} plus(Z_1,Y,Z) & \text{if } i = 1, \\ plus(Z_i,Y,Z_{i-1}),\ plus^{i-1}(Z_{i-1},Y,Z_{i-2}) & \text{if } i > 1. \end{cases}$$

Suppose the following query is posed:

query: *?-times(X,Y,Z), X=4, Z=48.*

Based on the compiled formula, we have the following terminating evaluation plans (see Figure 4-3). Notice that the finiteness constraints satisfied by *succ* and *plus* are: $fc = \{succ: 1 \rightarrow 2,\ succ: 2 \rightarrow 1,\ plus: \{1,2\} \rightarrow 3,\ plus: \{1,3\}\} \rightarrow 2,\ plus: \{2,3\}\} \rightarrow 1\}$.

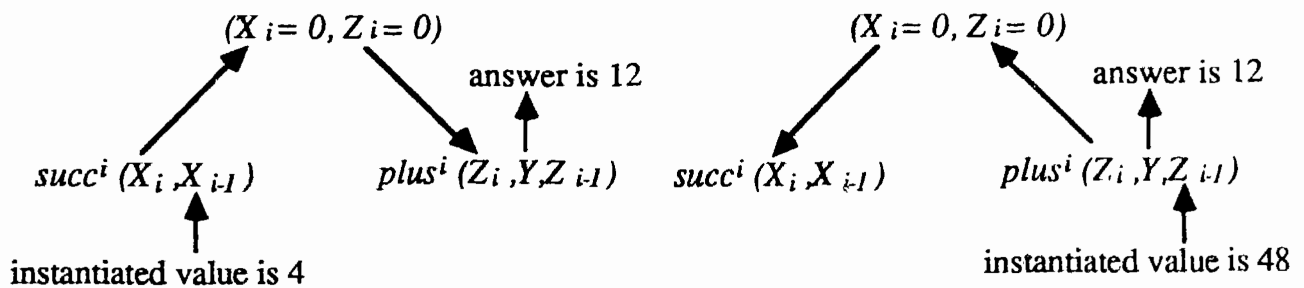


Figure 4-3 Two terminating evaluation plans for the *times* program

Evaluation plan 1:

Start the evaluation from the chain $succ^i(X_i, X_{i-1})$ to the exit rule $(X_i = 0, Z_i = 0)$ and then go to the chain $plus^i(Z_i, Y, Z_{i-1})$, that is, $succ^i(X_i, X_{i-1}) \Rightarrow (X_i = 0, Z_i = 0) \Rightarrow plus^i(Z_i, Y, Z_{i-1})$.

Evaluation plan 2:

Start the evaluation from the chain $plus^i(Z_i, Y, Z_{i-1})$ to the exit rule $(X_i = 0, Z_i = 0)$ and then go to the chain $succ^i(X_i, X_{i-1})$, that is, $plus^i(Z_i, Y, Z_{i-1}) \Rightarrow (X_i = 0, Z_i = 0) \Rightarrow succ^i(X_i, X_{i-1})$.

Careful readers will see that the first evaluation plan is more efficient than the second. In the next chapter we will discuss how to select an efficient evaluation plan.

Monotonicity constraints (mc's) have been recognized as useful in the analysis of termination [APPRSU89] [BroS89a] [BroS89b]. It is likely that a given query or a set of constraints provides some information which can be used to stop the evaluation along some chains.

Definition: A variable V in a compiled formula is **monotonic** if the value of V at the $(i+1)$th iteration is greater (or less) than the corresponding value at the ith iteration (for any $i > 0$), according to some partial order.

To capture the characteristic of monotonic variables and reflect the partial order explicitly, Han [Han91] proposed a mapping function ψ which maps an argument V to $\psi(V)$ and indicates the monotonicity of the argument. The mapping ψ can be an identity mapping, such as *Fare*, or from a nonnumerical value to a numerical one, such as *Length(L)*, where L is a list. After a proper mapping ψ is performed on a monotonic variable V, the following comparisons are valid:

$\psi(V) > c$, $\psi(V) \geq c$, $\psi(V) < c$, $\psi(V) \leq c$, or $\psi(V) = c$, where c is a constant in the domain of $\psi(V)$. The following lemma indicates how to identify monotonic variables.

Lemma 4-3: Suppose r is a compiled formula

$$r = e \cup \bigcup_{i=1}^{\infty} (p_1^i, p_2^i, \ldots, p_n^i, e)$$

and (s,t) is a chain invariant of p_j^i $(1 \leq j \leq n)$. If there is a monotonicity constraint p_j: $s <_{mc} t$, then the variables at positions s and t are monotonic.

Proof: According to Property 3-2, the variables at positions s and t share the same variable name between the kth chain element and $(k+1)$th chain element of p_j^i.

Without loss of generality, we only prove that the variable at position s is monotonic during the evaluation.

Let $value(k,s)$ denote the value of the variable at position s in the kth chain element of p_j^i. We prove that $value(k,s) <_{mc} value(k+1,s)$ for any k $(k \geq 0)$ during the evaluation.

According to the monotonicity constraint p_j: $s <_{mc} t$, $value(k,s) <_{mc} value(k,t)$ for any k $(k \geq 0)$. Since (s,t) is a chain invariant of p_j^i, $value(k,t) = value(k+1,s)$ for any k $(k \geq 0)$. Therefore, $value(k,s) <_{mc} value(k+1,s)$ for any k $(k \geq 0)$.

Thus the lemma holds.

□

To use monotonicity constraints, we present the following monotonicity blocking rules where the adornment β indicates that the monotonic variable is blocked[20] by some constant.

---

[20] A monotonic variable V is said to be **blocked** by some constant if there exists a mapping function ψ which maps V to a numerical value, and there exists a constant c such that one of the following conditions is satisfied:
a) $\psi(V) \leq c$ if the value of $\psi(V)$ monotonically increases, where $c < d$ if $\psi(V)$ is convergent to a limit d.
b) $\psi(V) \geq c$ if the value of $\psi(V)$ monotonically decreases, where $c > d$ if $\psi(V)$ is convergent to a limit d.

Monotonicity Blocking Rules: Suppose r is a compiled formula. When a query and a set of constraints are given, we adorn r as follows.

1. If a variable V of r is monotonic and a query constraint can block the growth of the value of V, then V is adorned with β, that is, V^{β}.
2. Two variables specified by "=" or with the same variable name are adorned with β if one of them is adorned with β.

4.5.1 Termination Detection

In this section, we present a sufficient condition under which the evaluation of compiled formulas is guaranteed to terminate. The algorithm for testing this condition is also presented.

Example 4-11: Consider the following program which defines the function $Z = X \ mod \ Y$.

$$mod(X,Y,Z) \text{ :- } X < Y,\ Z = X.$$
$$mod(X,Y,Z) \text{ :- } plus(X_1,Y,X),\ mod(X_1,Y,Z).$$

After the ith expansion, we have the following compiled formula:

$$mod(X,Y,Z) = (X < Y,\ Z = X) \cup \bigcup_{i=1}^{\infty} (plus^i(X_i,Y,X_{i-1}),\ (X_i < Y,\ Z = X_i))$$

where $plus^i(X_i,Y,X_{i-1})$ is defined in Example 4-10 with chain invariant (1,3) and satisfies the following monotonicity constraints: $mc = \{plus{:}\ 1 <_{mc} 3,\ plus{:}\ 2 <_{mc} 3\}$, where $<_{mc}$ refers to the usual order on integers. Therefore, according to lemma 4-3, the variables X and X_1 in the recursion are monotonic.

Suppose the following query is posed:

query: $?\text{-}mod(X,Y,Z),\ Y=2,\ Z=0,\ X \leq 8.$

The above observation is due to Han's work in [Han91].

Since X is a monotonic variable and there is a query constraint, $X \leq 8$, which can block the growth of X during the evaluation, we have the following finitely evaluable and terminating evaluation plan:

$$(X_i < Y, Z = X_i)$$
$$\downarrow$$
$$plus^i(X_i, Y, X_{i-1})$$

Figure 4-4 A finitely evaluable and terminating plan for the *mod* program.

<u>Evaluation plan:</u> $(X_i < Y, Z = X_i) \Rightarrow plus^i(X_i, Y, X_{i-1})$, that is, the evaluation starts from the exit rule $(X_i < Y, Z = X_i)$ and ends with the chain $plus^i(X_i, Y, X_{i-1})$.

By using the constraint $X \leq 8$, the evaluation can terminate at the fourth iteration with the solution $X=\{2,4,6,8\}$.

□

Theorem 4-3: Suppose r is a compiled formula. Given a query and a set of constraints, the evaluation of r terminates with respect to the query and constraints if the evaluation of any one chain terminates.

Proof: According to Property 3-1, all the chains of r are synchronized, that is, they all have the same expansion length during the evaluation. Therefore, the termination of the evaluation of one chain means the termination of the evaluation of all other chains.

□

Based on the above theorem and the monotonicity blocking rules, we give an algorithm to detect whether or not the evaluation of a compiled formula terminates with respect to a given query and a set of the constraints.

Algorithm 4-2: Termination Detection Algorithm.

INPUT: A finitely evaluable compiled formula, a query and a set of constraints.

OUTPUT: An assertion of whether the execution of the evaluation plan terminates ('yes') or not ('no') with respect to the query and the constraints.

METHOD:

If one of the following conditions is satisfied, then return 'yes'; otherwise, return 'no'.

(a) The query is an existence checking query and the existence checking condition is satisfied.

(b) The relation for every predicate of the compiled formula is finite with respect to the query and the constraints.

(c) The chain being evaluated is defined on a finite relation without cyclic data

(d) There is a monotonic variable V adorned with β after the monotonicity blocking rules are applied to the first expanded exit rule set.

□

4.5.2 Analysis of the Termination Detection Algorithm

Now we prove the correctness of the termination detection algorithm.

Proposition 4-2: The termination detection algorithm is correct in the sense that given a finitely evaluable compiled formula r, a query and a set of constraints, the evaluation of r can terminate if Algorithm 4-2 returns 'yes'.

Proof: Clearly, according to Theorem 4-3, we only need to show that if one of the conditions in Algorithm 4-2 is true, then there must exist one chain such that the evaluation along this chain terminates.

Condition (a): An existence checking query checks if the given fact exists in the database. When the existence of the fact is known (i.e., the existence checking condition is satisfied), there is no need to continue the evaluation along any chain and the entire evaluation should be stopped [Han89b].

Condition (b): If the relation for every predicate of the compiled formula is finite with respect to the query and the constraints, the evaluation must terminate when no new facts can be generated [BaRa86] [APPRSU89] [HanH89].

Condition (c): If the chain being evaluated is defined on a finite relation without cyclic data, the evaluation along this chain must terminate when no new facts can be generated.

Condition (d): If there is a monotonic variable V adorned with β after applying the monotonicity blocking rules to the first expanded exit rule set, it follows that V is a monotonic variable and the growth of the value of V will be blocked by some constant during the evaluation according to the monotonicity blocking rules. Therefore, the evaluation of the chain to which V belongs must terminate. According to Theorem 4-3, the evaluation of r terminates.

Thus Algorithm 4-2 is correct.

□

Example 4-12: Consider the following program which reverses a list Y to X.

reverse([], []).

$reverse([Z/X_1],Y)$:- $reverse(X_1,Y_1)$, $append(Y_1,Z,Y)$.

where $[Z/X_1]$ is the list construction function and $append(Y_1,Z,Y)$ is defined in Example 3-6.

After performing the FP-transformation and the rectification process, we have:

$reverse(X,Y)$:- $X = []$, $Y = []$.

$reverse(X,Y)$:- $cons(Z,X_1,X)$, $reverse(X_1,Y_1)$, $append(Y_1,Z,Y)$.

After the ith expansion, we have the following compiled formula:

$$reverse(X,Y) = (X = [\,], Y = [\,]) \cup \bigcup_{i=1}^{\infty} (cons^i(Z_{i-1},X_i,X_{i-1}), (X_i = [\,], Y_i = [\,]), append^i(Y_i,Z_{i-1},Y_{i-1}))$$

where

$$cons^i(Z_{i-1},X_i,X_{i-1}) = \begin{cases} cons(Z,X_1,X) & \text{if } i = 1, \\ cons(Z_{i-1},X_i,X_{i-1}),\ cons^{i-1}(Z_{i-2},X_{i-1},X_{i-2}) & \text{if } i > 1. \end{cases}$$

and

$$append^i(Y_i,Z_{i-1},Y_{i-1}) = \begin{cases} append(Y_1,Z,Y) & \text{if } i = 1, \\ append(Y_i,Z_{i-1},Y_{i-1}),\ append^{i-1}(Y_{i-1},Z_{i-2},Y_{i-2}) & \text{if } i > 1. \end{cases}$$

Suppose the following queries are posed:

query 1: *?-reverse(X,Y), X=[a,b,c].*

query 2: *?-reverse(X,Y), Y=[a,b,c].*

query 3: *?-reverse(X,Y).*

For query 1 one possible evaluation plan is: $cons^i(Z_{i-1},X_i,X_{i-1}) \Rightarrow (X_i = [\,], Y_i = [\,]) \Rightarrow append^i(Y_i, Z_{i-1},Y_{i-1})$. For query 2 one possible evaluation plan is: $append^i(Y_i, Z_{i-1},Y_{i-1}) \Rightarrow (X_i = [\,], Y_i = [\,]) \Rightarrow cons^i(Z_{i-1},X_i,X_{i-1})$. However, for query 3 there is no finitely evaluable and terminating evaluation plan. This example also indicates that the evaluation of a compiled formula should start at the finitely evaluable and terminating chains.

□

Whether or not an evaluation plan for a compiled formula is safe is influenced by whether or not sufficient constraints are provided. In the next chapter, we will discuss how to generate a safe, constraint-based evaluation plan for a compiled formula.

Chapter 5

Constraint-Based Evaluation of Compiled Formulas

The preceding chapter discusses the *conditions* which guarantee the safety of compiled formulas. This chapter discusses the *evaluation* of compiled formulas by incorporating constraints.

Three classes of constraints are discussed: integrity constraints, rule constraints and query constraints. Three types of integrity constraints are considered: type constraints, finiteness constraints and monotonicity constraints.

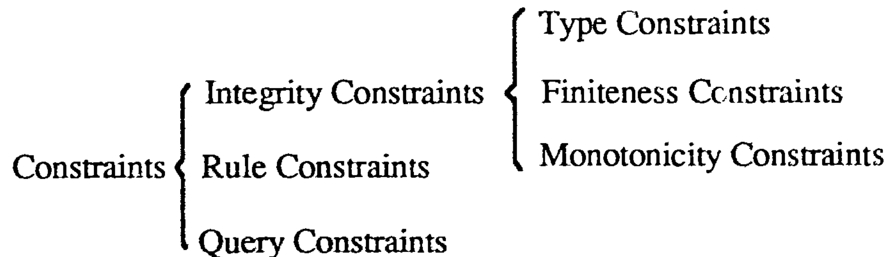


Figure 5-1 Constraints used in the evaluation of compiled formulas.

Integrity constraints essentially reflect the relationship among data in databases and/or the arguments of functions and predicates in deduction rules. Type constraints are used to check the type compatibility between the arguments of predicates and the instantiated information provided by queries. Finiteness constraints are used to generate finitely evaluable plans. Generating a finitely evaluable plan for a compiled formula means ordering the chains and the exit rule set of the compiled formula in such a way that the finite evaluability can be guaranteed. An algorithm is presented to generate the finitely evaluable plan for a compiled formula. Monotonicity constraints are used in termination detection.

Rule constraints have been recognized to be useful in search space reduction during the evaluation of compiled formulas [HEH89] [Han91]. We will show that rule constraints should be compiled together with the transformed and rectified functional recursions to reduce the search space in iterative processing.

Query constraints are also shown to be useful in search space reduction. They can be used to select evaluation directions and to restrict the search space during the evaluation. By selection of evaluation directions, we mean to choose some chain(s) or the exit rule set to start the evaluation. The main selection criterion is selectivity. It has been recognized that an efficient evaluation of recursions should start from a small set which utilizes all the binding information [BaRa86] [HanH87] [Han89b] [Han88a] [Ullm85] [BeRa87]. One example is the magic set method whose basic idea is based on this criterion [BMSU86] [SaZa87] [MFP90] [BeRa87]. It uses the binding information provided by queries to derive a small set, called *magic set*, to start the evaluation. Our study shows that query constraints are also very important in deriving a small set to start the evaluation. Moreover, query constraints can also be used *during the evaluation* of recursions to restrict the search space.

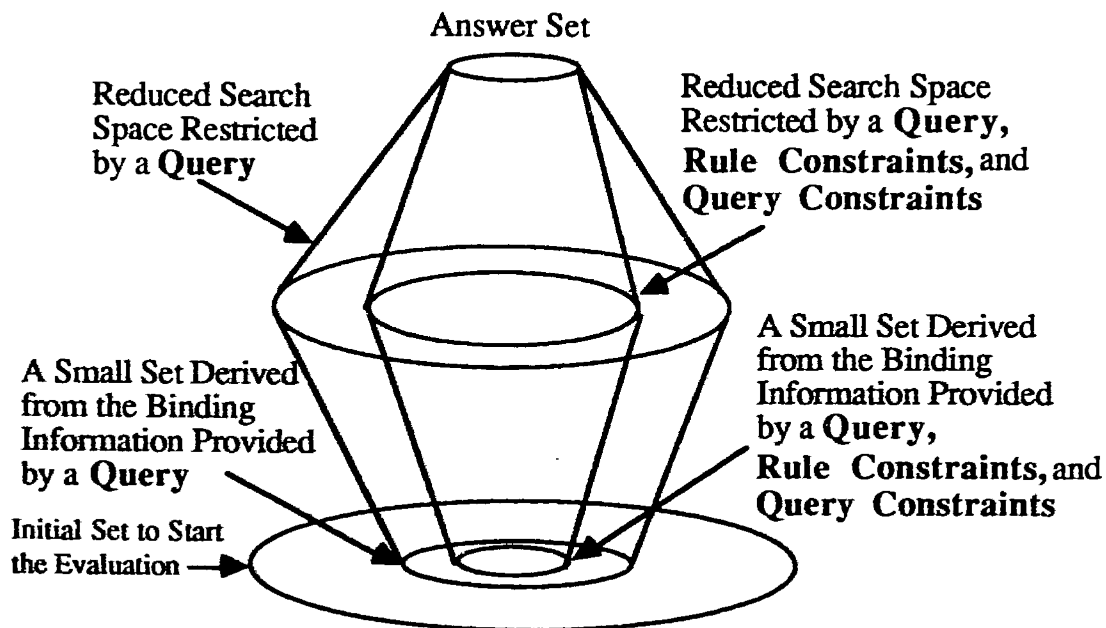


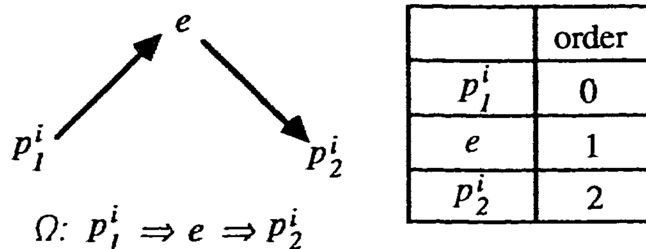
Figure 5-2 Search space reduction by using rule constraints and query constraints.

In our study, a general model for cost is assumed. Many general cost models (also called measuring models or measuring functions in the literature [CGM90] [ZHH89] [KrZa88]) have been developed for database systems. Since our evaluation method is independent of the cost model used, we merely summarize some properties of cost models. The cost is usually dependent on parameters, such as the join methods used, access methods available, disk I/O, cardinality of the operands, selectivities, etc. The cost influences the selection of efficient evaluation plans.

Suppose r is a compiled formula:

$$r = e \cup \bigcup_{i=1}^{\infty} (p_1^i, p_2^i, \ldots, p_n^i, e).$$

Let $\Omega: \Pi_0 \Rightarrow \Pi_1 \Rightarrow \ldots \Rightarrow \Pi_k$ denote an evaluation plan for r, where Ω is the plan name and $\Pi_i = \{q \mid q$ is a chain or the exit rule set of $r\}$ $(1 \leq i \leq k \leq n)$[21]. $\Pi_i \Rightarrow \Pi_j$ means that every predicate in Π_i will be evaluated before any predicate in Π_j is evaluated $(0 \leq i \leq k, 0 \leq j \leq k)$. Each predicate q in Π_i is associated with an integer i, called the **evaluation order** of the predicate and denoted by $order(q)$. In other words, $order(q) = i$ iff $q \in \Pi_i$. For instance, if $n = 2$ and $\Omega: p_1^i \Rightarrow e \Rightarrow p_2^i$ is a safe evaluation plan, then such a plan and the evaluation order can be illustrated as follows.

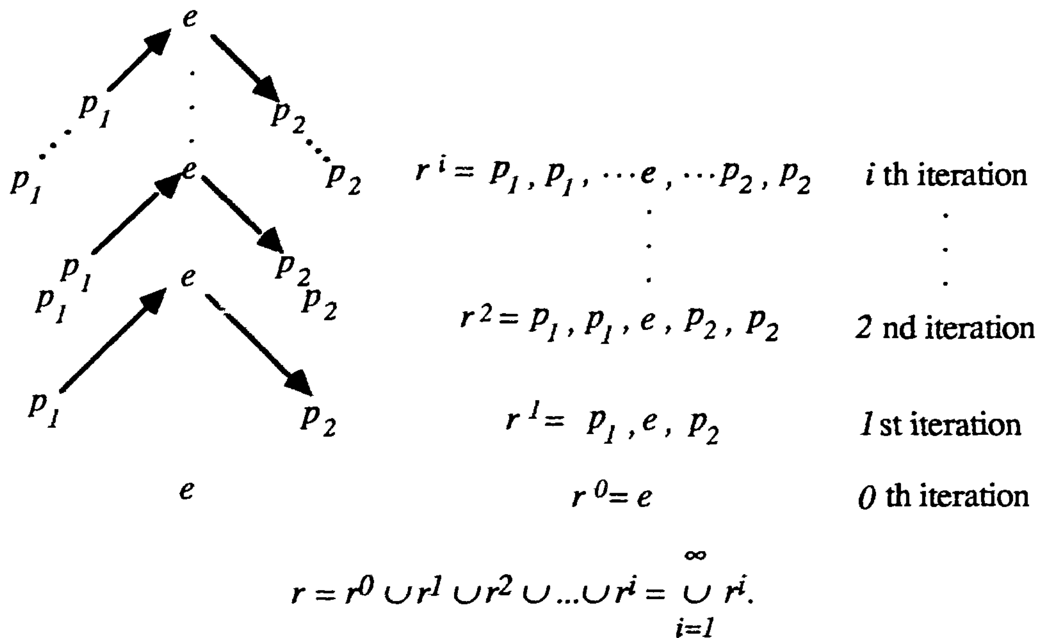


	order
p_1^i	0
e	1
p_2^i	2

Figure 5-3 Illustration of evaluation plans and evaluation orders.

[21] k is a positive integer and is less than or equal to the total number of the chains of r.

In essence, the evaluation of a compiled formula with respect to a given query and a set of constraints is based on the relational operations, particularly join and union operations [BaRa86] [Ullm89b]. Therefore, when a query and a set of constraints are given, the evaluation of the compiled formula $r = e \cup \bigcup_{i=1}^{\infty} (p_1^i, p_2^i, e)$ according to Ω: $p_1^i \Rightarrow e \Rightarrow p_2^i$ can be illustrated as follows.



$$r = r^0 \cup r^1 \cup r^2 \cup ... \cup r^i = \bigcup_{i=1}^{\infty} r^i.$$

Figure 5-4 Illustration of the evaluation of a compiled formula.

As we can see from Figure 5-4, query constraints can be used in the following ways: (i) *before the evaluation*, they can be used to select a small, relevant set (i.e., the cost is relatively small) to start the evaluation; (ii) *during the evaluation*, they can be used to reduce the search space to a more relevant one by discarding those facts which do not satisfy the constraints; (iii) *at the end of the evaluation*, they should be used to select the satisfactory answers.

The generation of an efficient evaluation plan for a compiled formula is affected by two factors: the evaluation plan must be safe, and the evaluation should start from the predicate(s) with relatively small cost.

This chapter is organized as follows. Section 5.1 discusses the use of integrity constraints, with each subsection concentrating on one type of integrity constraint. The generation of safe evaluation plans is considered. Section 5.2 discusses the incorporation of rule constraints to reduce the search space. Section 5.3 discusses how to use query constraints to reduce the search space. In Section 5.4, we present a method to generate a safe, constraint-based evaluation plan for a compiled formula.

5.1 Incorporation of Integrity Constraints

In this section, we consider three types of integrity constraints and discuss how to incorporate them into the evaluation of a compiled formula.

5.1.1 Incorporation of Type Constraints

As discussed in Section 4.2, the types of predicate arguments are registered as type constraints which should be used to check the type compatibility between the arguments of predicates and the instantiated information provided by queries. The following example shows the importance of type checking.

Example 5-1: Consider the following *natural_number* program discussed in Example 3-4.

$natural_number(X) :- X = 0.$

$natural_number(X) :- natural_number(X_1), succ(X_1,X).$

According to the results, we have the following compiled formula:

$$natural_number(X) \;=\; (X = 0) \bigcup_{i=1}^{\infty} ((X_i = 0),\; succ^i(X_i,X_{i-1})).$$

The following query asks if 4.5 is a natural number.

query: *?-natural_number(X), X = 4.5.*

Because all the arguments of *succ* are defined on integers but 4.5 is not an integer, this query should be rejected.

□

When a database is defined, the types of attributes of EDB relations are defined. The types of arguments of the functions supported by the database are also defined. When an IDB predicate is defined, the types of its arguments can be inferred.

Let V_p denote a variable of predicate p. We say the type of V_p can be inferred from that of V_q , denoted by $V_p \leftarrow V_q$, if there is a rule with p as the head and q in the body, or there exists a q' where $V_p \leftarrow V_{q'}$, $V_{q'} \leftarrow V_q$ (transitivity).

Type Inference Rules:

Suppose V_p is a variable of predicate p.

1. If p is an EDB predicate or a functional predicate[22], the type of V_p is already defined when the database is defined.
2. If p is an IDB predicate and $V_p \leftarrow V_q$, the type of V_p is that of V_q.

When an IDB predicate is defined, the type inference rules should be applied to the IDB predicate and the types of its variables should be registered as type constraints. Using type constraints, we can design an algorithm to check type compatibility.

---

[22] Note that a functional predicate is transformed from a function symbol and the FP-transformation associates all the integrity constraints (including type constraints) specified for the function with the relation for the functional predicate.

Algorithm 5-1: Type Compatibility Checking Algorithm.

INPUT: A query predicate and the type constraints for the predicate.

OUTPUT: An assertion of whether every instantiated value in the query satisfies the type constraints.

METHOD:

if any value provided by the query violates the type constraints registered for the predicate
  then return('no')
  else return('yes').

□

Example 5-2: Consider the following program which computes the greatest common divisor of two integers.

$gcd(X,0,X).$

$gcd(X,X_1,Z)$:- $mod(X,X_1,X_2),\ gcd(X_1,X_2,Z).$

query: $?\text{-}gcd(X,X_1,Z),\ X = a,\ X_1 = b.$

where $mod(X,X_1,X_2)$ is defined in Example 4-11 and the query asks what is the greatest common divisor of a and b.

The type compatibility checking algorithm returns 'no' since a and b are characters which do not match the types of the arguments of the IDB predicate gcd.

□

5.1.2 Incorporation of Finiteness Constraints

Section 4.4 gives a necessary and sufficient condition which guarantees the finite evaluability of a compiled formula. In this section, we discuss how to use finiteness constraints to generate finitely evaluable plans.

Suppose r is a compiled formula and Ω: $\Pi_0 \Rightarrow \Pi_1 \Rightarrow \ldots \Rightarrow \Pi_k$ is an evaluation plan for r. Ω is said to be a *finitely evaluable plan* if every relation for the predicate being evaluated according to Ω is finite.

The existence of a finitely evaluable plan depends on whether the finite evaluability testing algorithm (Algorithm 4-1) returns 'yes'. Therefore, when a query and a set of constraints are given, the finite evaluability testing algorithm should be performed first.

The discussion in Chapter 4 indicates that the finitely evaluable plans for a compiled formula r should be generated according to the finiteness propagation in r. To generate a finitely evaluable order for a compiled formula, we use a finiteness dependency graph to analyze the finiteness propagation.

Definition: Suppose r is a compiled formula. The **finiteness dependency graph** of r with respect to a given query and a set of constraints is a directed graph $G=(U,E)$, where $U = \{ \upsilon \mid \upsilon$ is a chain or the exit rule set of $r \}$ and $(\upsilon,\mu) \in E$ *iff* the finiteness of one variable of μ is propagated from υ via a shared variable.

We examine an example below.

Example 5-3: Consider the following *append* program discussed in Example 3-6.

$append(U,V,W)$:- $U = []$, $V = W$.

$append(U,V,W)$:- $cons(X_1,U_1,U)$, $append(U_1,V,W_1)$, $cons(X_1,W_1,W)$.

query: $?\text{-}append(U,V,W),\ W = [a,b,c]$.

where the functional predicate *cons* satisfies the finiteness constraints $fc = \{cons: \{1,2\} \rightarrow 3, cons: 3 \rightarrow \{1,2\}\}$ (Example 4-4).

According to the results (Example 3-6), we have the following compiled formula:

$$append(U,V,W) = (U=[], V=W) \cup \bigcup_{i=1}^{\infty} (cons^i(X_i,U_i,U_{i-1}), (U_i=[], V=W_i), cons^i(X_i,W_i,W_{i-1})).$$

The finiteness propagation in the above rule is shown below.

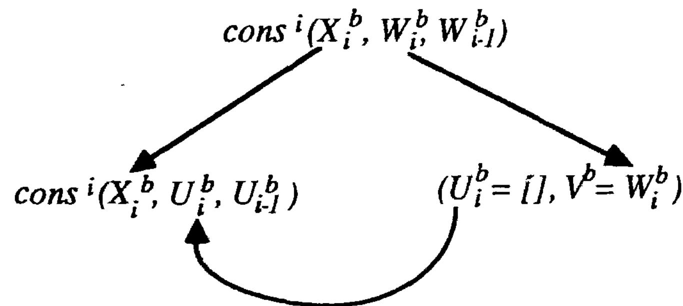


Figure 5-5 The finiteness propagation in the *append* program.

□

Generation of Finitely Evaluable Plans

Based on the finiteness propagation in a compiled formula, we can develop an algorithm to generate a finitely evaluable plan.

Algorithm 5-2: Finitely Evaluable Plan Generating Algorithm.

INPUT: A compiled formula, a query and a set of constraints.

OUTPUT: A finitely evaluable plan in which the chains and the exit rule set are assigned an evaluation order, or 'no' if there is no finitely evaluable plan.

METHOD:

1. Test the finite evaluability of the compiled formula (Algorithm 4-1). If the compiled formula is not finitely evaluable, stop and inform the user with 'no'.
2. For each node[23] υ in the finiteness dependency graph of the compiled formula, *order*(υ):=*0*.
3. Find all the strongly connected components (SCCs) and replace each SCC with a supernode (S_node). Let *DAG* denote the new, acyclic graph.
4. Perform a topological sort on *DAG* by increasing the evaluation order by *1* whenever an arc is encountered.
5. Every node in each SCC is assigned the same evaluation order as that of the corresponding S_node.

□

Notice that if all the predicates of a compiled formula are EDB predicates, Algorithm 5-2 will assign the same evaluation order, *0*, to each of them. In other words, a finite set of answers can always be generated if the relations for these predicates are finite. Several researchers have drawn the same conclusion [BaRa86] [APPRSU89] [SaVa89] [Ullm89a].

Before analyzing Algorithm 5-2, we examine an example.

[23] Notice that a node corresponds to a chain or the exit rule set of a compiled formula.

Example 5-4: The following finitely evaluable order is generated by applying Algorithm 5-1 to the *append* program discussed in Example 5-3 with respect to the following query:

query: *?-append(U,V,W), W = [a,b,c].*

and finiteness constraints: $fc = \{cons: \{1,2\} \rightarrow 3,\ cons: 3 \rightarrow \{1,2\}\}$.

	order
$cons^i(X_i,W_i,W_{i-1})$	0
$(U_i^b=[],\ V^b = W_i^b)$	1
$cons^i(X_i,U_i,U_{i-1})$	2

Table 5-1 The evaluation order generated for the *append* program.

Therefore, a finitely evaluable plan is: $cons^i(X_i,W_i,W_{i-1}) \Rightarrow (U_i=[],\ V = W_i) \Rightarrow cons^i(X_i,U_i,U_{i-1})$.

□

Analysis of the Finitely Evaluable Plan Generating Algorithm

Now we analyze Algorithm 5-2 and show that it can order the chains and the exit rule set of a compiled formula.

Theorem 5-1: Algorithm 5-2 is correct in the sense that given a compiled formula, a query and a set of constraints, the evaluation plan generated by Algorithm 5-2 is finitely evaluable.

Proof:

Step 1 of Algorithm 5-2 ensures that there exists a finitely evaluable plan and its correctness is guaranteed by Theorem 4-1. Step 2 is an initialization step which ensures that every

node has an evaluation order. Step 3 generates a directed, acyclic graph (*DAG*) in which each S_node represents a set of strongly connected nodes. The topological sort performed by step 4 increases the evaluation order by *1* whenever an arc is encountered. This is correct because the node whose finiteness is propagated from another predicate must be delayed until it becomes finitely evaluable. Step 5 is correct because the nodes whose finiteness depends on each other (i.e., they are strongly connected) should be evaluated together (have the same evaluation order as that of the corresponding S-node).

Thus the theorem holds.

□

Algorithm 5-2 is linear because finding all SCCs of a directed graph $G=(V,E)$ can be done in $O(max(|V|, |E|))$ time [AHU74] and a topological sort on a $DAG=(V',E')$ can be done in $O(|V'|+|E'|)$ time [AHU74]. The intuitive meaning of Algorithm 5-2 is that the evaluation of the predicates whose finiteness is dependent on other predicates should be delayed until their relations become finite.

Example 5-5: We use another query to examine the *gcd* program discussed in Example 5-2.

$gcd(X,0,X).$

$gcd(X,X_1,Z)$:- $mod(X,X_1,X_2)$, $gcd(X_1,X_2,Z)$.

where $mod(X,X_1,X_2)$ is defined in Example 4-11.

After performing the FP-transformation and the rectification process (see Sections 3.1 and 3.2), we have:

$gcd(X,X_1,Z)$:- $X_1 = 0, Z = X$.

$gcd(X,X_1,Z)$:- $mod(X,X_1,X_2)$, $gcd(X_1,X_2,Z)$.

After the ith expansion, we have the following compiled formula:

$$gcd(X,X_1,Z) = (X_1 = 0, Z = X) \cup \bigcup_{i=1}^{\infty} (mod^i(X_{i-1},X_i,X_{i+1}), (X_{i+1} = 0, Z = X_i)).$$

where

$$mod^i(X_{i-1},X_i,X_{i+1}) = \begin{cases} mod(X,X_1,X_2) & \text{if } i = 1, \\ mod(X_{i-1},X_i,X_{i+1}),\ mod^{i-1}(X_{i-2},X_{i-1},X_i) & \text{if } i > 1. \end{cases}$$

Suppose the following query is posed:

query: $?\text{-}gcd(X,X_1,Z),\ X = 8,\ Z = 2.$

The finiteness propagation is illustrated as follows, where a finiteness constraint $mod: \{1,2\} \rightarrow 3$ is used.

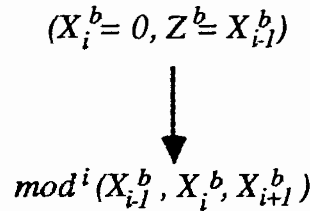


Figure 5-6 The finiteness propagation in the *gcd* program.

After performing Algorithm 5-2, the following evaluation order is generated:

	order
$(X_i = 0, Z = X_{i-1})$	*0*
$mod^i(X_{i-1},X_i,X_{i+1})$	*1*

Table 5-2 The evaluation order generated for the *gcd* program.

Therefore, a finitely evaluation plan is: $(X_i = 0, Z = X_{i-1}) \Rightarrow mod^i(X_{i-1},X_i,X_{i+1})$.

□

5.1.3 Incorporation of Monotonicity Constraints

Section 4.5 gives a sufficient condition which guarantees the termination of the evaluation of a compiled formula. In this section, we enhance the results developed in Section 4.5 and demonstrate the application of monotonicity constraints in termination detection.

As shown in Algorithm 4-2, the evaluation of a compiled formula in which functional predicates are defined on finite domains always terminates because the functional predicates obtained from the FP-transformation can be viewed as finite EDB relations [HanW90] [Han91]. In addition, the evaluation also terminates if some constraints restrict every infinite domain to a finite one. However, if the relation for a predicate being evaluated is defined on an infinite domain or can not be restricted to a finite domain by any constraint, it is often essential to apply monotonicity constraints to terminate the evaluation.

Example 5-6: We examine the application of monotonicity constraints.

1. Arithmetic Functions

   An arithmetic operation often implies the monotonicity of a function. For instance, $S_1 > 0, Fare_1 > 0, and\ S_1 + Fare_1 = Fare$ imply that $Fare > S_1$ and $Fare > Fare_1$. $X_1 = succ(X)$ implies $X_1 > X$.

2. List Functions

   The monotonicity of a list manipulation function usually lies at the growing or shrinking of the length of the list. For example, *cons* results in a longer list than the original ones.

3. Term Construction Functions

   The monotonicity of a term constructor/de-constructor is similar to that of a list operation. The repetitive application of a term constructor results in an increasingly deeply-nested

sequence of functors, such as $f(f(...f(X)...))$, while the repetitive application of a term de-constructor, such as f^{-1}, results in an increasingly less-nested sequence of functors. In other words, the nested level has monotonic behavior.

4. EDB Relations

   Some attributes in an EDB relation may have certain monotonic behavior. For example, the arrival time of a flight is always later than its departure time. An acyclic EDB relation can also be viewed as a partially ordered finite relation.

5. Termination of Iterations

   One may like to terminate the evaluation of a compiled formula after a certain number of iterations when there is no appropriate termination condition. It essentially treats the number of iterations as a monotonically increasing function.

□

Lemma 4-3 indicates how to identify monotonic variables. Han [Han91] introduced a mapping function ψ which maps the value of a variable V to a numerical value $\psi(V)$ and he gave the following result: if the value of $\psi(V)$ at the $(i+1)$th iteration is greater (or less) than the corresponding value at the ith iteration (for any $i > 0$), then $\psi(V)$ is monotonic [Han91]. The constant which blocks the growth of $\psi(V)$ is called a *termination restraint* [HanW90] [Han91]. A termination restraint can be provided by a query, a constraint, a rule, or an EDB relation. If $\psi(V)$ is convergent to a limit, say d, the termination constraint should be less (greater) than d if $\psi(V)$ is monotonically increasing (decreasing) during the evaluation. The monotonicity blocking rules (Section 4.5) are based on the above observations.

According to the monotonicity constraints shown in Section 4.5, a variable V is adorned with β (i.e., V^{β}) if it is a monotonic variable and the growth of its values is blocked by some constant during the evaluation. If we use the mapping function ψ introduced in [Han91], the

above rule can be elaborated as follows. Suppose the values of $\psi(V)$ monotonically *increase*. If there is a constant c provided by a query or a constraint which validates the following comparisons: $\psi(V) < c$, $\psi(V) = c$, or $\psi(V) \leq c$ ($c < d$ if $\psi(V)$ is convergent to a limit d), V should be adorned with β, indicating that the growth of the value of V is blocked during the evaluation. A similar interpretation can be given if the values of $\psi(V)$ monotonically *decreases*.

Example 5-7: In Example 3-7, the value of *Fare* in the IDB predicate *travel* is monotonically increasing but not convergent to a limit. Thus a restraint on the maximum value of *Fare*, such as $Fare \leq 500$, ensures the termination of the evaluation. Therefore, *Fare* should be adorned with β (i.e., $Fare^{\beta}$) and we should register *500* as a termination restraint of *Fare* so that we can use this information to terminate the evaluation.

Similarly, the length of L in the predicate *travel* is monotonically increasing since $length(L) > length(L_1)$. Thus a restraint on the maximum length of L, such as, $length(L) < 7$, ensures the termination of the evaluation and L should be adorned with β. In addition, we should register the constant 7 as a termination restraint on L.

□

The following algorithm is used to register the possible termination restraints for those variables which have monotonic behavior according to some partial order, say ψ.

Algorithm 5-3: Incorporation of Monotonicity Constraints.

INPUT: A finitely evaluable plan, a query and a set of constraints.

OUTPUT: A safe evaluable plan in which termination restraints are registered.

METHOD:

1. Apply the termination detection algorithm (Algorithm 4-2). If it returns 'no', stop and inform the user with 'no'.

2. If there exists a variable V such that V is adorned with β, register the constant c which blocks the growth of V as a termination restraint.

□

The above algorithm enhances Algorithm 4-2 by registering the termination restraint(s) so that we can use this information to terminate the evaluation. Notice that if there exists a variable V adorned with β, there must exist a constant which can block the growth of V (see the monotonicity blocking rules in Section 4.5).

5.2 Incorporation of Rule Constraints

In this section, we discuss how to use rule constraints to reduce the search space.

As discussed in Section 4.2, a rule constraint adds one or more conjuncts to the body of a deduction rule and changes it to a new, more constrained one. Therefore, compilation should be performed on the modified set of deduction rules. Such a modification confines the search to a smaller EDB relations or enforces stronger constraints on the join of EDB relations, and thus reduces the search space. We examine an example below.

Example 5-8: Consider the following rule constraints added to the *travel* program.

(a) *international airport constraint:* The intermediate airports must be international airports.

(b) *lay-over time constraint:* The lay-over time should be between one and three hours.

(c) *same flight direction constraint:* The flight direction of each connecting flight should be the same as that of the entire travel.

The above constraints can be added to the body of the recursive rule of the *travel* program. For example,

constraint (a) can be added to the rule as *international*(Dep_1), where Dep_1 represents an intermediate airport (Example 4-6).
constraint (b) can be added to the rule as $1 \leq IDTime - IATime, IDTime - IATime \leq 3$, and constraint (c) can be added to the rule as *same_direction*(Dep, Dep_1, Arr).

By adding these constraints to the body of the deduction rule, the original rule becomes a new, more constrained one. For example, *international*(Dep_1) confines the search for transfer airports to international airports only and reduces the size of the EDB relation to be participated in iterative processing. The other two constraints play similar roles.

□

Notice that a powerful system should allow rule constraints to be added and/or deleted flexibly. A method for dynamic association of deduction rules with rule constraints is studied in [HEH89].

Now we show that incorporating rule constraints into the query independent compilation can reduce the search space [Han91]. Consider the following single-chain recursion:

$$r(X) :- p(Y), r(Z).$$

where X, Y and Z are variable vectors and p is a chain element. Suppose $c_1(V_1)$, ..., and $c_i(V_i)$ are rule constraints. By adding them to the body of the rule, the modified rule becomes,

$$r(X) :- p(Y), r(Z), c_1(V_1), ..., c_i(V_i).$$

The newly added constraints form conjunctions with the predicates p and r. Since they enforce restrictions on the deduction rule, if they are compiled together with the rule, stronger constraints are enforced on the rule, which reduces the size of the relation for r generated during the iterative processing and the cost of the evaluation.

The case of a multi-chain rule can be reasoned similarly.

Therefore, we have the following results [Han91]:

If a rule constraint enforces restrictions on a set of deduction rules in a functional linear recursion, the compilation of rule constraints together with the corresponding rules reduces the cost of query evaluation.

Rule constraints can be used to constrain the *nondistinguished variables* (the variables appearing only in the body of the rule) of a deduction rule. However, these variables cannot be constrained at query level because they do not appear in any query predicate. Some user requirements, though seemingly like query constraints, are essentially rule constraints. For example, a traveler may require that (i) the lay-over time between each pair of connecting flights be less than two hours, or (ii) every intermediate airport must be an international airport. Such constraints must include some nondistinguished variable(s) in the body of the recursive rule and cannot be specified at the query level. Therefore they should be rule constraints, which add constraints to the body of the recursive rule rather than to the query predicate.

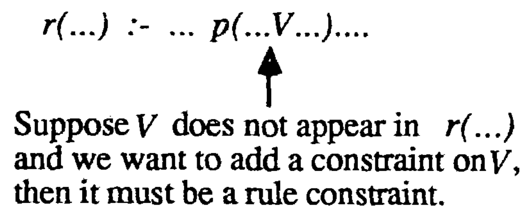


Figure 5-7 Rule constraints added on a nondistinguished variable.

5.3 Incorporation of Query Constraints

A popular heuristic of query processing in both traditional and deductive database systems is *pushing selection as deeply as possible into a relational expression* [Han91] [Ullm89b]. Since a

selection is often represented by a query constraint, the heuristic can be rephrased as **pushing query constraints as deeply as possible into the compiled formulas in the query evaluation.**

However, it is inappropriate to *blindly* push all the query constraints into a compiled chain in the evaluation. Techniques should be developed for appropriate use of query constraints in the processing.

Assume that $r(X_0)$ is a query predicate and $C_1(X_1)$, $C_2(X_2)$, ..., $C_i(X_i)$ are query constraints, where X_0, ..., X_i are variable vectors. That is, a query is of the form,

$$?\text{-}r(X_0),\ C_1(X_1),\ C_2(X_2),\ ...,\ C_i(X_i).$$

The above query constraints are used in the following ways to reduce the search space: (i) *Before the evaluation*, they can be treated as query instantiation information used at the start of a chain processing. To reduce the size of the initial relation at the start of the iterative processing, a more selective end should be taken as the start point. (ii) *During the evaluation*, they should be used to reduce the search space to a more relevant one by discarding those facts which do not satisfy the constraints. (iii) *At the end of the evaluation*, they should be used to select satisfactory answers.

We examine an example.

Example 5-9: Suppose a user poses a query on the *travel* database as follows:

find a set of (connecting) flights from Vancouver to Ottawa, departing after 8am, arriving between 6pm and 7pm, with the total fare no more than $500.

The above request can be expressed by the following query:

query: ?-travel(L,Dep,DTime,Arr,ATime,Fare).

with the query constraints:

(a) $Dep = vancouver$,

(b) $Arr = ottawa$,

(c) $DTime \geq 8$,

(d) $ATime \geq 18$,

(e) $ATime \leq 19$,

(f) $Fare \leq 500$.

Before the evaluation, a more relevant, selective predicate should be chosen to start the evaluation. Therefore, the evaluation should start at the arrival end because the query constraints *(d)* and *(e)* provide better selective information for the arrival end than the departure end (i.e., query constraint *(c)*).

The constraint *(f)*, $Fare \leq 500$, should be used *during the evaluation* and applied at each iteration. Since the value of *Fare*, defined by the function *sum*, is monotonically increasing, the query constraint *(f)* can be used to reduce the search space based on the following observation: if the value of *Fare* generated at an iteration is greater than *500*, the final answer generated from this intermediate result will not be a satisfactory answer.

To derive satisfactory answers, the query constraint *(c)* should be used *at the end of the evaluation* to ensure that the departure time is after *8am*.

□

5.4 Generation of Safe, Constraint-Based Evaluation Plans

As we can see from the previous sections, a constraint can be used in the analysis of safety and/or in search space reduction. For example, $Fare \leq 500$ can be used to terminate the evaluation of the *travel* program and to reduce the search space during the evaluation. In this section, we discuss how to use different constraints to generate a safe, constraint-based evaluation plan for a compiled formula. The efficiency is achieved by pushing constraints into the evaluation to reduce the search space.

Even though a general model for cost is assumed in our study, the cost is normally affected by different implementation methods and is usually dependent on many parameters, (e.g., the join methods used, access methods available, disk I/O, cardinality of the operands, selectivities, etc.). Since the cost is in most cases system-dependent, the actual cost is normally considered as a black box [CGM90] except for those properties which can be utilized in the query processing [KrZa88]. One property used by several methods (e.g., the magic set method [BMSU86], the counting method [SaZa86], the multi-way counting method [Han89b], the magic counting methods [SaZa87]) is selectivity. In this thesis, we mainly consider this property.

In practice (e.g., LDL prototype [TsZa86] [CGKNTZ90]), the cost of an unsafe evaluation is considered as an infinite cost [KrZa88]. Therefore, safe evaluation must be guaranteed first. When evaluation is guaranteed to be safe, the selectivity should be considered by using the given query and set of constraints.

In the evaluation of a chain (e.g., $succ^i(X_i,X_{i-1})$), processing can proceed in two different directions: ***forward evaluation*** which proceeds from the ith chain element to the $(i+1)$th element (e.g., from $succ(X_i,X_{i-1})$ to $succ(X_{i+1},X_i)$), and ***backward evaluation*** which proceeds in the reverse direction (e.g., from $succ(X_{i+1},X_i)$ to $succ(X_i,X_{i-1})$).

The first criterion in the selection of an evaluation direction is safety. For example, for the chain $succ^i(X_i,X_{i-1})$, if the finiteness of X_i can only be propagated from X_{i-1}, the evaluation direction must be forward. When both directions can guarantee safety, the selection criterion becomes the selectivity of each end. For instance, in Example 5-9, the evaluation of the *flight* chain can start from either end: start the search from the departure end (*vancouver*) or the arrival end (*ottawa*). However, as analyzed in Example 5-9, the evaluation from the arrival end is more efficient because the query constraints *(d)* $ATime \geq 18$ and *(e)* $ATime \leq 19$ provide more selective information for the arrival end than the information provided by the query constraint *(c)* $DTime \geq 8$ for the departure end.

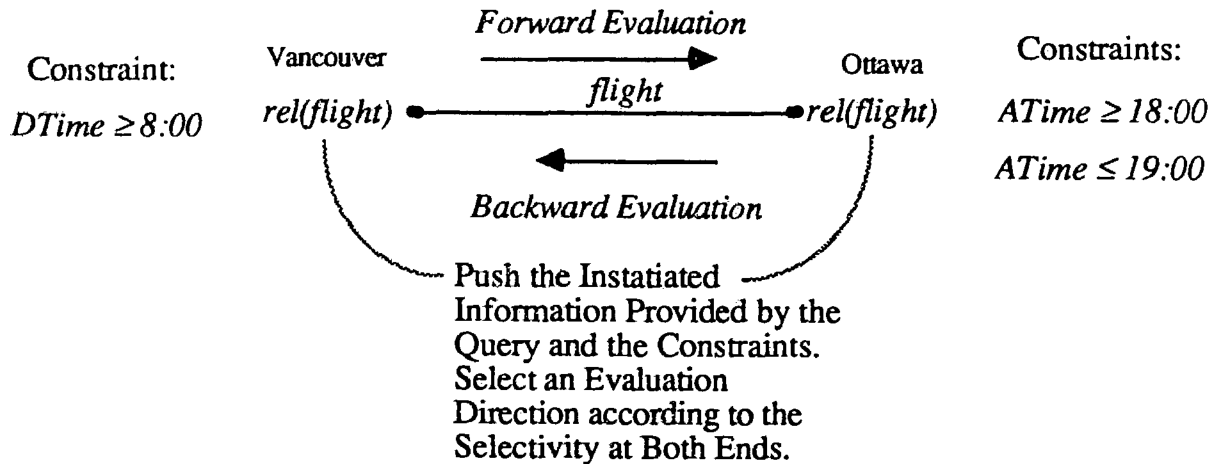


Figure 5-8 Selection of evaluation directions according to selectivity.

We also notice that there may be more than one way to propagate finiteness in a compiled formula. In this case, we should propagate the finiteness from predicate(s) with relatively small cost. We examine an example below.

Example 5-10: Recall that in Example 4-10 we have two evaluation plans when query *?-times(4,Y,48)* is posed:

$$\Omega_1: succ^i(X_i,X_{i-1}) \Rightarrow (X_i = 0,\ Z_i = 0) \Rightarrow plus^i(Z_i,Y,Z_{i-1}).$$

$$\Omega_2: plus^i(Z_i,Y,Z_{i-1}) \Rightarrow (X_i = 0,\ Z_i = 0) \Rightarrow succ^i(X_i,X_{i-1}).$$

As discussed in Example 4-10, both evaluation plans Ω_1 and Ω_2 are safe. However, finiteness should be propagated according to Ω_1 because the relation generated by $succ^i(X_i,X_{i-1})$ is much smaller than the relation generated by $plus^i(Z_i,Y,Z_{i-1})$. In order words, finiteness should be propagated from the predicate with relatively small cost (i.e., $succ^i(X_i,X_{i-1})$).

□

Algorithm 5-4: Generation of an Efficient Evaluation Plan for a Compiled Formula.

INPUT: A compiled formula, a query and a set of constraints.

OUTPUT: A safe, constraint-based evaluation plan or 'no' if there is no safe plan.

METHOD:

1. Apply Algorithm 5-1 to check the type compatibility. If Algorithm 5-1 returns 'no', stop and inform the user with 'no'.
2. Use the instantiated information provided by the query and the constraints to estimate the relative cost of evaluating each chain and the exit rule set.
3. Based on the cost analysis, propagate the finiteness from the predicate(s) with relatively small cost. If the evaluation of a chain can start from forward and backward directions, determine one evaluation direction based on the relative selectivity of the query constraints at both ends of the chain. Apply the query constraints belonging to this end as query instantiations to reduce the size of the initial set.
4. Apply Algorithm 5-2 to generate a finitely evaluable plan. If Algorithm 5-2 returns 'no', stop and inform the user with 'no'.
5. Apply Algorithm 5-3 to detect if the plan is a terminating evaluation plan and register the termination restraint(s). If it is not a terminating plan, stop and inform the user with 'no'.
6. Use the information provided by the query, the rule constraints and query constraints to discard those facts which do not satisfy the requirement of the query or the constraints.

7. Use termination restraint(s) to terminate the evaluation when necessary. When the evaluation terminates with a set of answers, select those answers which satisfy the query constraints.

□

Theorem 5-2: Algorithm 5-4 is correct in the sense that, given a compiled formula, a query and a set of constraints, the evaluation plan generated by Algorithm 5-4 is a safe evaluation plan and correctly incorporates the given constraints.

Proof:

Step 1 uses type constraints and is necessary since invalid information provided by a query is inconsistent with a database system. Steps 2 and 3 ensure that finiteness is propagated from the predicate(s) with relatively small cost so that the finite evaluation plan generated by step 4 can start from a more relevant and more selective set. To estimate the cost, it is necessary to use the instantiated information provided by the query and different types of constraints. Step 4 ensures the finite evaluability of the evaluation plan. Its correctness is guaranteed by Theorem 5-1. Step 5 ensures that the evaluation must terminate. Its correctness is guaranteed by Theorem 4-2. Step 6 utilizes the information provided by the query, the rule constraints and the query constraints in the evaluation to reduce the search space. Any fact which does not satisfy this information should be discarded since the final answer derived from this fact will not be a satisfactory answer. Step 7 is necessary since Step 2 already ensures the termination of the evaluation. If there is a variable which has monotonic behavior during the evaluation, the termination restraint(s) should be used to terminate the evaluation when possible. In addition, only those answers which satisfy the query constraints should be selected.

□

To outline the results developed in this chapter, we examine the *travel* program discussed in Chapter 1.

Example 5-11: According to the results in Example 3-7, we have the following compiled formula.

$$travel(L,Dep,DTime,Arr,ATime,Fare) = flight(L,Dep,DTime,Arr,ATime,Fare) \cup \bigcup_{i=1}^{\infty} (flight^i(Fno_{i-1},Dep_{i-1},DTime_{i-1},Dep_i,IATime_{i-1},S_{i-1}), flight(L_i,Dep_i,IDTime_{i-1},Arr,ATime,Fare_i), sum^i(S_{i-1},Fare_i,Fare_{i-1}), cons^i(Fno_{i-1},L_i,L_{i-1})).$$

where $flight^i(Fno_{i-1},Dep_{i-1},DTime_{i-1},Dep_i,IATime_{i-1},S_{i-1})$, $sum^i(S_{i-1},Fare_i,Fare_{i-1})$, and $cons^i(Fno_{i-1},L_i,L_{i-1})$ are defined in Example 3-7.

We analyze the questions asked by the customer in Example 1-1 and discuss safe, constraint-based evaluation plans to answer these questions. The queries corresponding to these questions are from Examples 2-5 and 4-6.

query 1: ***?-travel(L,Dep,_,Arr,_,_), Dep = vancouver, Arr = ottawa.***

The finiteness propagation with respect to the above query can be illustrated as follows.

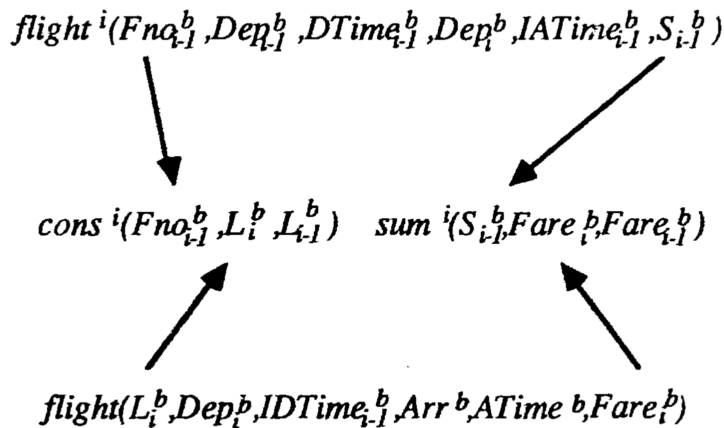


Figure 5-9 The finiteness propagation in the *travel* program.

After performing Algorithm 5-2, we have the following evaluation order:

	order
$flight(L_i,Dep_i,IDTime_{i-1},Arr,ATime,Fare_i)$	*0*
$flight^i(Fno_{i-1},Dep_{i-1},DTime_{i-1},Dep_i,IATime_{i-1},S_{i-1})$	*0*
$cons^i(Fno_{i-1},L_i,L_{i-1})$	*1*
$sum^i(S_{i-1},Fare_i,Fare_{i-1})$	*1*

Table 5-3 The evaluation order generated for the *travel* program.

Therefore, the finitely evaluable plan is:

$$\{flight(L_i,Dep_i,IDTime_{i-1},Arr,ATime,Fare_i), flight^i(Fno_{i-1},Dep_{i-1},DTime_{i-1},Dep_i,IATime_{i-1},S_{i-1})\} \Rightarrow \{cons^i(Fno_{i-1},L_i,L_{i-1}),\ sum^i(S_{i-1},Fare_i,Fare_{i-1})\}.$$

After performing Algorithm 5-3, a 'yes' is returned, indicating that the above evaluation plan can terminate.

The above evaluation plan indicates that the evaluation can start at either end: the *Vancouver* end or the *Ottawa* end.

query 2: ***?-travel(L,Dep,DTime,Arr,_,_),***

Dep = vancouver, Arr = ottawa, 7:00 ≤ DTime, DTime ≤ 8:00.

Interested readers can see that the safe evaluation plan for the above query is the same as that for query 1:

$$\{flight(L_i,Dep_i,IDTime_{i-1},Arr,ATime,Fare_i), flight^i(Fno_{i-1},Dep_{i-1},DTime_{i-1},Dep_i,IATime_{i-1},S_{i-1})\} \Rightarrow \{cons^i(Fno_{i-1},L_i,L_{i-1}),\ sum^i(S_{i-1},Fare_i,Fare_{i-1})\}.$$

However, the query constraints $7{:}00 \leq DTime$ and $DTime \leq 8{:}00$ provide better selective information, that is, the cost for evaluating from the *Vancouver* end will be lower than that from the *Ottawa* end. Therefore, the evaluation should start from the *Vancouver* end and the evaluation plan should be:

$flight^i(Fno_{i-1},Dep_{i-1},DTime_{i-1},Dep_i,IATime_{i-1},S_{i-1}) \Rightarrow flight(L_i,Dep_i,IDTime_{i-1},Arr,ATime,Fare_i)$ $\Rightarrow \{cons^i(Fno_{i-1},L_i,L_{i-1}),\ sum^i(S_{i-1},Fare_i,Fare_{i-1})\}$.

query 3: ***?-travel(L,Dep,_,Arr,ATime,_),***

Dep = vancouver, Arr = ottawa, 11:45 < ATime, ATime < 12:15.

Interested readers can examine that the safe evaluation plan for the above query is the same as that for query 1:

However, the query constraints $11{:}45 < ATime$ and $ATime < 12{:}15$ provide better selective information, the cost for evaluating from the *Ottawa* end will be lower than that from the *Vancouver* end. Therefore, the evaluation should start from the *Ottawa* end and the evaluation plan should be:

$flight(L_i,Dep_i,IDTime_{i-1},Arr,ATime,Fare_i) \Rightarrow flight^i(Fno_{i-1},Dep_{i-1},DTime_{i-1},Dep_i,IATime_{i-1},S_{i-1}) \Rightarrow \{cons^i(Fno_{i-1},L_i,L_{i-1}),\ sum^i(S_{i-1},Fare_i,Fare_{i-1})\}$.

query 4: ***?-travel(L,Dep,_,Arr,_,Fare), Dep = vancouver, Arr = ottawa,***

Fare ≤ 500.

Interested readers can examine that the safe evaluation plan for the above query is also the same as that for query 1:

However, the information provided by the query constraint $Fare \leq 500$ can be used during the evaluation to reduce the search space. We notice that *Fare* is a monotonic variable

(monotonically increasing) during the evaluation. Therefore, whenever an intermediate answer with *Fare* greater than *500* is generated, this answer should be discarded since any result derived from this fact will not be a satisfactory answer to the query. Consequently, the search space can be restricted to a smaller, more relevant portion.

query 5: ***?-travel(L,Dep,_,Arr,_,_), Dep = vancouver, Arr = ottawa, Fare ≤ 500 and every intermediate airport must be an international airport..***

Recall that to express the above constraint, we need a rule constraint, *international*(Dep_1), which registers all the international airport names in its relation. As analyzed in Section 5.2, this rule constraint should be incorporated into the compilation.

Let $int_flight(Fno,Dep,DTime,Dep_1,IATime,S)$:- $flight(Fno,Dep,DTime,Dep_1,IATime,S)$, $\textbf{international}(Dep_1)$

where the relation for the rule constraint *international*(Dep_1) registers all the international airports.

Then we have the following compiled formula:

$$travel(L,Dep,DTime,Arr,ATime,Fare) = flight(L,Dep,DTime,Arr,ATime,Fare) \cup \bigcup_{i=1}^{\infty} (int_flight^i(Fno_{i-1},Dep_{i-1},DTime_{i-1},Dep_i,IATime_{i-1},S_{i-1}), flight(L_i,Dep_i,IDTime_{i-1},Arr,ATime,Fare_i), sum^i(S_{i-1},Fare_i,Fare_{i-1}), cons^i(Fno_{i-1},L_i,L_{i-1})).$$

Notice that the relation for *int_flight* now becomes a more constrained one which contains only the information about those flights which stop at international airports. Therefore, any intermediate answer which does not satisfy this rule constraint should be discarded during the evaluation. Consequently, the search space can be restricted to a smaller, more relevant portion.

□

Chapter 6

Conclusion

We have examined the evaluation of functional recursions. Since functions are normally defined on infinite domains, safety is treated as an important issue in this study. Different meanings of safety used in the literature were discussed and our definition of safety was presented. To facilitate the analysis of safety, constraints were categorized into three classes: query constraints, rule constraints and integrity constraints. We showed that for compiled formulas, safety can be viewed as a combination of two properties: finite evaluability and termination. A necessary and sufficient condition guaranteeing the finite evaluability of a compiled formula was presented. The algorithm for testing the condition was developed. A sufficient condition guaranteeing the termination of the evaluation of a compiled formula was presented. The algorithm for testing that condition was also developed.

The generation of efficient evaluation plans is another important aspect of this study. We proposed a safe, constraint-based evaluation method for compiled formulas. Based on the classification of constraints, we showed that type constraints should be used to check type compatibility. Finiteness constraints can be used to generate finitely evaluable plans based on finiteness dependency graphs and a finiteness ordering algorithm developed in this thesis. Monotonicity constraints can be used in termination detection. Rule constraints should be used in compilation to reduce the search space. Query constraints are shown to be useful in the selection of efficient evaluation plans and search space reduction.

Safety and the efficient evaluation of functional recursions have been investigated in previous studies [BMSU86] [BroS89b] [KRS88] [RBS87] [SaVa89] which pave the way for our

research. Regarding the safety of functional recursions, the difference between our study and previous research is that we consider safety with respect to a given query and a set of constraints specified over a given database instance. In addition, our study of safety is based on compiled formulas. This approach allows us to a give detailed analysis of safety. Regarding the efficient evaluation of functional recursions, our study is based on the query-independent compilation method. This allows us to incorporate constraints into the recursive query processing and generate efficient evaluation plans.

However, the functional recursions studied in this thesis are limited to linear recursions. It is important to extend our results to other recursions, such as non-linear recursions and mutual recursions. If a non-linear recursion or a mutual recursion can be transformed into a linear recursion, the techniques developed in this thesis can be applied; otherwise, new techniques might be required.

References

[AG88] S. Abiteboul and S. Grumbach, "COL: a Logic-Based Language for Complex Objects", Proceedings of International Conference on Extending Database Technology (EDBT'88), Venice, Italy, March 1988, pp. 271-293.

[AH88] S. Abiteboul and R. Hull, "Data Functions, Datalog and Negation", Proceedings of the 1988 ACM-SIGMOD Conference on Management of Data, Chicago, IL, 1988, pp. 143-153.

[AHU74] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1984.

[APPRSU89] F. Afrati, C. H. Papadimitriou, G. Papageorgiou, A. Roussou, Y. Sagiv and J. D. Ullman, "On the Convergence of Query Evaluation", *Journal of Computer and System Sciences*, Vol. 38 No. 2, 1989, pp. 341-359.

[BaRa86] F. Bancilhon and R. Ramakrishnan, "An Amateur's Introduction to Recursive Query Processing Strategies", Proceedings of the 1986 ACM-SIGMOD Conference on Management of Data, Washington, D.C., May 1986, pp. 16-52.

[BeLe86] M. Bellia and G. Levi, "Relation Between Logic and Functional Languages: A Survey", *Journal of Logic Programming*, Vol. 3, No. 2, 1986, pp. 217-236.

[BeRa87] C. Beeri and R. Ramakrishnan, "On the Power of Magic", Proceedings of the 6th ACM Symposium on Principles of Database Systems, San Diego, CA, March 1987, pp. 269-283.

[BMSU86] F. Bancilhon, D. Maier, Y. Sagiv and J. D. Ullman, "Magic Sets and Other Strange Ways to Implement Logic Programs", Proceedings of the 5th ACM Symposium on Principles of Database Systems, Cambridge, MA, March 1986, pp. 1-15.

[BNRST87] C. Beeri, S. Naqvi, R. Ramakrishnan, O. Shmueli and S. Tsur, "Sets and Negation in a Logic Database Language (LDL1)", Proceedings of the 6th ACM Symposium on Principles of Database Systems, San Diego, CA, March 1987, pp. 21-37.

[BroS89a] A. Brodsky and Y. Sagiv, "Inference of Monotonicity Constraints in Datalog Programs", Proceedings of the 8th ACM Symposium on Principles of Database Systems, Philadelphia, PA, March 1989, pp. 190-199.

[BroS89b] A. Brodsky and Y. Sagiv, "On Termination of Datalog Programs", Proceedings of the 1st International Conference on Deductive and Object-Oriented Databases (DOOD'89), Kyoto, Japan, December 1989, pp. 95-112.

[CGKNTZ90] D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur and C. Zaniolo, "The LDL System Prototype", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 1, 1990, pp. 76-90.

[CGM90] U. S. Chakravarthy, J. Grant and J. Minker, "Logic-Based Approach to Semantic Query Optimization", *ACM Transactions on Database Systems*, Vol. 15, No. 2, 1990, pp. 162-207.

[Coh90] J. Cohen, "Constraint Logic Programming Languages", *Communications of the ACM*, Vol. 33, No. 7, 1990, pp. 52-68.

[Col90] A. Colmerauer, "An Introduction to PROLOG III", *Communications of the ACM*, Vol. 33, No. 7, 1990, pp. 69-90.

[Date90] C. J. Date, *An Introduction to Database Systems*, Vol. 1, 5th Edition, Addison-Wesley, Reading, MA, 1990.

[DL86] D. DeGroot and G. Lindstrom, *Logic Programming—Functions, Relations, and Equations*, Prentice-Hall, Englewood, NJ, 1986.

[GMN84] H. Gallaire, J. Minker and J. Nicolas, "Logic and Databases: A Deductive Approach", *ACM Computing Surveys*, Vol. 16, No. 2, 1984, pp. 153-185.

[Gru89] S. Grumbach, "Integration of Functions Defined with Rewriting Rules in Datalog", Proceedings of the 1st International Conference on Deductive and Object-Oriented Databases (DOOD'89), Kyoto, Japan, December 1989, pp. 317-335.

[Han88a] J. Han, "Selection of Processing Strategies for Different Recursive Queries", Proceedings of the 3rd International Conference on Data and Knowledge Bases, Jerusalem, Israel, June 1988, pp. 59-68.

[Han88b] J. Han, "Single and Multi-Chain Recursion: The Core of General Linear Recursion", SFU CSS/LCCR Technical Report TR88-3, Simon Fraser University, Burnaby, B.C., Canada, February 1988.

[Han89a] J. Han, "Compiling General Linear Recursions by Variable Connection Graph Analysis", *Computational Intelligence*, Vol. 5, No. 1, 1989, pp. 12-31.

[Han89b] J. Han, "Multi-Way Counting Method", *Information Systems*, Vol. 14, No. 3, 1989, pp. 219-229.

[Han91] J. Han, "Constraint-Based Reasoning in Deductive Databases", Proceedings of the 7th International Conference on Data Engineering, Kobe, Japan, April, 1991.

[HanH87] J. Han and L. J. Henschen, "Handling Redundancy in the Processing of Recursive Database Queries", Proceedings of the 1987 ACM-SIGMOD Conference on Management of Data, San Fransisco, CA, May 1987, pp. 73-81.

[HanH89] J. Han and L. J. Henschen, "The Level-Cycle Merging Method", Proceedings of the 1st International Conference on Deductive and Object-Oriented Databases (DOOD'89), Kyoto, Japan, December 1989, pp. 113-129.

[HanHY88] J. Han, L. J. Henschen and C. Youn, "Compiling Complex Linear Recursive Clusters", Proceedings of the 1988 CIPS Conference, Edmonton, Alberta, November 1988, pp. 101-110.

[HanHZ89] J. Han, L. J. Henschen and N. Zhuang, "Derivation of Magic Sets by Compilation", Proceedings of the 1989 International Conference on Software Engineering and Knowledge Engineering, Chicago, IL, June 1989, pp. 164-171.

[HanW90] J. Han and Q. Wang, "Efficient Evaluation of Functional Single Linear Recursions in Deductive Databases", SFU CSS/LCCR Technical Report TR90-2, Simon Fraser University, Burnaby, B.C., Canada, February 1990.

[HEH89] Jianing Han, D. Epley and Jiawei Han, "Compiling Search Constraints for Deductive and Recursive Databases", Proceedings of the 2nd International Symposium on Artificial Intelligence, Monterrey, Mexico, October 1989.

[Ioan85] Y. E. Ioannidis, "A Time Bound on the Materialization of Some Recursively Defined Views", Proceedings of the 11th International Conference on Very Large Data Bases, Stockholm, Sweden, August 1985, pp. 219-226.

[Kif88] M. Kifer, "On Safety, Domain Independence and Capturability of Database Queries", Proceedings of the 3rd International Conference on Data and Knowledge Bases, Jerusalem, Israel, June 1988, pp. 405-415.

[KiL88] M. Kifer and E. L. Lozinskii, "SYGRAF: Implementing Logic Programs in a Database Style", *IEEE Transactions on Software Engineering*, Vol. 14, No. 7, 1988.

[KiRS88] M. Kifer, R. Ramakrishnan and A. Siberschatz, "An Axiomatic Approach to Deciding Query Safety in Deductive Databases", Proceedings of the 7the ACM Symposium on Principles of Database Systems, Austin, TX, 1988, pp. 52-60.

[KKR90] P. C. Kanellakis, G. M. Kuper and P. Z. Revesz, "Constraint Query Languages", Proceedings of the 9th ACM Symposium on Principles of Database Systems, Nashville, TN, April 1990, pp. 299-313.

[KRS88] R. Krishnamurthy, R. Ramakrishnan and O. Shmueli, "A Framework for Testing Safety and Effective Computability of Extended Datalog", Proceedings of the 7th ACM Symposium on Principles of Database Systems, Austin, TX, March 1988, pp. 154-163.

[KrZa88] R. Krishnamurthy and C. Zaniolo, "Optimization in a Logic Based Language for Knowledge and Data Intensive Applications", Proceedings of International Conference of Extending Database Technology (EDBT'88), Venice, Italy, March 1988, pp. 16-33.

[Las90] Jean-Louis Lassez, "Querying Constraints", Proceedings of the 9th ACM Symposium on Principles of Database Systems, Nashville, TN, April 1990, pp. 288-298.

[MaW88] D. Maier and D.S. Warren, *Computing with Logic: Logic Programming with Prolog*, Benjamin Cummings, Menlo Park, CA, 1988.

[MFP90] I. S. Mumick, S. J. Finkelstein and H. Pirahesh, "Magic Conditions", Proceedings of the 9th ACM Symposium on Principles of Database Systems, Nashville, TN, April 1990, pp. 314-329.

[MNSUV87] K. Morris, J. F. Naughton, Y. Saraiya, J. D. Ullman and A. V. Gelder, "An Overview of the NAIL! System", *Data Engineering*, Vol. 10, No. 4, 1987, pp. 28-43.

[PS87] F. C. N. Pereira and S. M. Shieber, *Prolog and Natural-Language Analysis*, Center for the Study of Language and Information (CSLI), Menlo Park, CA, 1987.

[RBS87] R. Ramakrishnan, F. Bançilhon and A. Silberschatz, "Safety of Recursive Horn Clauses with Infinite Relations", Proceedings of the 6th ACM Symposium on Principles of Database Systems, San Diego, CA, March 1987, pp. 328-339.

[SaVa89] Y. Sagiv and M. Vardi, "Safety of Datalog Queries over Infinite Databases", Proceedings of the 8th ACM Symposium on Principles of Database Systems, Philadelphia, PA, March 1989, pp. 160-171.

[SaZa86] D. Sacca and C. Zaniolo, "The Generalized Counting Method for Recursive Queries", Proceedings of the 1st International Conference on Database Theory, Rome, Italy, 1986, pp. 31-53.

[SaZa87] D. Sacca and C. Zaniolo, "Magic Counting Methods", Proceedings of the 1987 ACM-SIGMOD Conference on Management of Data, San Fransisco, CA, May 1987, pp. 49-59.

[Shm87] O. Shmueli, "Decidability and Expressiveness Aspects of Logic Queries", Proceedings of the 6th ACM Symposium on Principles of Database Systems, San Diego, CA, March 1987, pp. 237-249.

[StSh86] L. Sterling and E. Shapiro, *The Art of Prolog*, The MIT Press, Cambridge, MA, 1986.

[STZ88] O. Shmueli, S. Tsur and C. Zaniolo, "Rewriting of Rules Containing Set Terms in a Logic Data Language (LDL)", Proceedings of the 7th ACM Symposium on Principles of Database Systems, Austin, TX, March 1988, pp. 15-28.

[SuHen90] A. Al-Sukairi and L. J. Henschen, "Query Indepedent Compilation of Linear Recursions", Proceedings of 1990 International Conference on Software Engineering and Knowledge Engineering, Chicago, IL, June 1990.

[TsZa86] S. Tsur and C. Zaniolo, "LDL: A Logic-Based Data-Language", Proceedings of the 12th International Conference on Very Large Data Bases, Kyoto, Japan, August 1986, pp. 33-41.

[Ullm85] J. D. Ullman, "Implementation of Logical Query Languages for Databases", *ACM Transactions on Database Systems*, Vol. 10, No. 3, 1985, pp. 289-321.

[Ullm89a] J. D. Ullman, "Bottom-up Beats Top-down for Datalog", Proceedings of the 8th ACM Symposium on Principles of Database Systems, Philadelphia, PA, March 1989, pp. 140-149.

[Ullm89b] J. D. Ullman, *Principles of Database and Knowledge-Base Systems*, Vols. 1 and 2, Computer Science Press, Rockville, MD, 1989.

[YHH88] C. Youn, L. J. Henschen and J. Han, "Classification of Recursive Formulas in Deductive Databases", Proceedings of the 1988 ACM-SIGMOD Conference on Management of Data, Chicago, IL, June 1988, pp. 320-328.

[Zani86] C. Zaniolo, "Safety and Compilation of Non-Recursive Horn Clauses", Proceedings of the 1st International Conference on Expert Database Systems, Charleston, SC, 1986, pp. 167-178.

[ZHH89] N. Zhuang, L. J. Henschen and J. Han, "Complexity Analysis and Performance Evaluation of Methods for Processing Linear Recursive Database Queries", Proceedings of the 1st Conference of International Association of Knowledge Engineers, College Park, MD, June 1989, pp. 334-342.