COMPILATION AND EVALUATION OF NESTED LINEAR RECURSIONS: A DEDUCTIVE DATABASE APPROACH

by

Tong Lu

B.S., East China Normal University, Shanghai, China, 1990

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
November 1993

APPROVAL

Name:	Tong Lu
Degree:	Master of Science
Title of Thesis:	Compilation and Evaluation of Nested Linear Recursions: A Deductive Database Approach
Examining Committee:	Dr. Tiko Kameda Professor Computing Science, Chairman
	Dr. Jiawei Han, Senior Supervisor
	Dr. William Havens, Supervisor
	Dr. Ze-Nian Li, Examiner
Date Approved:	November 15, 1993

Abstract

A deductive database system is an extension of a relational database system by supporting a rule-based, more expressive database language while preserving the set-oriented and declarative style of a relational database query language.

This thesis studies the implementation and extension of the chain-based compilation and evaluation method, an interesting method for deductive query evaluation.

Our work makes the following two contributions : (1) a query-independent compilation method is developed in C using Lex/Yacc, which automatically generates compiled chain-forms for linear recursions; and (2) the applicable domain of the chain-based compilation and evaluation method is extended to functional nested linear recursions.

The query-independent compilation method is based on the expansion regularity of a graph matrix, the V-matrix, which represents the variable connection pattern of a recursive rule. A complex linear recursion can be compiled into a highly regular chain-form and linear normal form, which facilitates efficient query analysis and processing.

The study on the extension of the applicable domain of the chain-based compilation and evaluation method to functional nested linear recursions leads to the systematic analysis of a typical logic program, the n-queens program. Our analysis shows that nested linear recursions can be implemented systematically and efficiently using the chain-based query evaluation method.

Acknowledgments

My greatest thanks go to professor Jiawei Han, my senior supervisor, for inspiring my interest in deductive databases, for his responsiveness, encouragement and considerations. The important work of this thesis comes from the written works and many discussions with him. Thanks also go to professor William Havens and professor Zenian Li, for their valuable comments and suggestions.

I would also like to express my gratitude to the following people: to professor Fuling Huang, my teacher in China, for her warm encouragement; to professor Woshun Luk, for his interesting classes which aroused my interest in database systems; to Hong Fan and Wei Lu, for their friendship and thoughtful advice when I needed them most; to Zhongmin Guo, Ju Wu, Zhaohui Xie and Ling Liu, for some helpful discussions on the project; to Alan Brewer and Bhavesh Doshi, for their careful job of proofreading; and to my friends, fellow students and colleagues: Armin Bruderlin, Ben Yao, Biaodong Cai, Beysim Segzin, James McDaniel, Jeff Dailey, Langlee Chu, Lingyun Wang, Martin Vorbeck, Natalie Shannon, Hong Shi, Jun Yu, Qiang Wang, Wei Zhou, Yifeng Zheng, Yongjian Fu, Youmin Zhang and others, for their friendship and help. Special thanks go to Ms.Kersti Jaager, for her kind help throughout my graduate years.

Finally, I would like to thank my parents and sister, for their encouragement and support in all these years. They have always let me know for sure that their love will be with me wherever I am.

Contents

List of Figures

Chapter 1
Introduction

This Chapter provides a general introduction of the Deductive Database System and an overview of this thesis.

1.1 Motivations

The research in Deductive Database Systems started in the late 1970's with a summary in the book "Logic and Databases" by Gallaire and Minker. From the beginning, research in this area has been motivated by two converging trends[BaRa86]: (i) the desire to integrate database technology and artificial intelligence technology, to extend database systems and provide them with the functionality of expert systems, thus creating "the knowledge base systems", and (ii) the desire to integrate logic programming technology and database technology and extend the power of the interface of the database system to that of a general purpose programming language. The feasibility of this integration is based on the fact that logic programming and relational calculus have the same underlying mathematical model : *the first-order logic*. Since the mid 1980's, substantial research has been done on deductive database systems, with the focus on *query optimization*, an essential issue on building high performance systems for large applications.

Over the past decade, relational database system has gained enormous popularity due to its uniformed data structure, its successful implementation of data management functions (such as integrity, data sharing and recovery), and its query language SQL, which is a declarative and set-oriented language for both data definition and management. However, there are a number of applications that have a database flavor and yet are not well-addressed by conventional Database Management Systems. These new applications pose demands that are not answered by conventional DBMS, such as the need to deal with complex structures , recursively defined objects and more flexible queries [UllZa90] .

Deductive Database Systems are thus systems which extend relational database systems while preserving their programming style by supporting a rule-based language capable of expressing complete applications. These systems are based on predicate logic as a data model, following a Prolog style of using Horn-clause or if-then rules to define predicates or relations. A salient feature of deductive database systems is their capability of supporting a declarative, rule-based style of expressing queries and applications on large databases.

1.2 Logic Database : EDB + IDB + Query

Let's start with a simple example and look at the components of a "logic database".

Example 1-1. A simple Logic Database.

Suppose we have two relations: *person(name)* and *parent(child, parent)* stored in a relational database. The instances of these relations are as follows [CeGoLe89] :

person: name
Anne
Bertrand
Charles
Dorothy
Evelyn
Fred
George
Hilary

parent: child	parent
Dorothy	George
Evelyn	George
Betrand	Dorothy
Anne	Dorothy
Anne	Hilary
Charles	Evelyn

These relations express the set of *ground facts*, just like the facts we specify as ground clauses in Prolog programs. The difference is that, in Prolog, all the knowledge (facts and rules) relevant to a particular application is contained within a single logic program P, while in Deductive Databases, we are dealing with a large number of facts which must be manipulated by a database system for its efficient data management functionality, such as data retrieving, data modification, data sharing, concurrency and consistency control.

In Deductive Databases, we call the set of ground facts **Extensional Database**(EDB) , which should be physically stored in a relational database.

Moreover, we have a set of *rules* which specify derived relationships based on the data stored in EDB. For example, with the above EDB storing two base relations (or tables), we can define a derived relation sg(for same_generation) in a **Datalog** program **P** as:

sg(X, X) :- person(X). (r_1)

sg(X,Y) :- parent(X, X_1), sg(X_1, Y_1), parent(Y, Y_1). (r_2)

Due to rule r_1, the derived relation "sg" contains a tuple <p,p> for each individual in the "person" relation. The rule r_2 is recursive and states that two persons are same generation cousins if their parents are same generation cousins.

Thus, based on the current data in the relations "person" and "parent", we can derive the relation "sg" as: < Anne, Anne >, < Anne, Betrand >, <Anne, Charles >, < Betrand, Anne >, < Betrand, Betrand >, < Betrand, Charles >, ..., <George, George >.

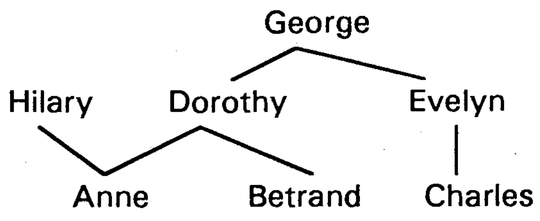


Figure 1-1. The Family Tree of example 1-1.

In Deductive Databases, we call the set of derived relations **Intensional Database** (IDB). The tuples or facts of an IDB relation is not physically stored in mass memory as EDB facts, but are derived from the current EDB data upon query.

The IDB relations derived from the EDB and a set of deduction rules are usually very large. However, it often happens that a user is only interested in a subset of these relations. For instance, they might only want to know the same generation cousins of "Anne" rather than all same generation cousins of all the people in the EDB.

To express such additional constraints, we can specify a *query* to the IDB. A **query** is a single literal preceded by a question mark and a dash, for example "?- sg(anne, X)". Queries serve to formulate ad-hoc requests against a "view" defined by a set of derivative rules. □

Usually the EDB is considered as a time-varying collection of information while the rule set is a time-invariant mapping which associates an IDB to each possible database state. In this sense, IDB relations correspond to relational views, which is a form of supporting a restricted set of logic queries. The significant difference between IDB definitions and relational views is that views are restricted to non-recursive definitions while the definition of an IDB relation can be recursive. The efficient evaluation of recursive queries is an important task of Deductive Database research.

1.3 Datalog = Database + Prolog ??

The language for Deductive Databases is **Datalog**. Datalog defines the *syntax* and *semantics* of relations in Deductive Database (like SQL to Relational Database Systems), it has been designed and intensively studied over the last few years and has been well accepted as the fundamental language of Deductive Database Systems. The efficient implementation and further extensions to Datalog is the current focus of DDB research.

From the syntax point of view, Datalog is in many respects a simplified version of a general logic programming language, such as Prolog. It consists of a finite set of **facts** and **rules.** Facts are assertions about a relevant piece of the world while rules are statements which allow us to deduce facts from other facts. Both *facts* and *rules* are particular forms of *knowledge*.

In the formalism of Datalog, both facts and rules are represented as **Horn Clauses** of the form L_0 :- L_1, ..., L_n. Each L_i is a **literal** of the form $P_i(t_1, \dots t_k)$ such that P_i is a **predicate** symbol and t_j's are **terms**. A term is either a constant or a variable.

The left hand side (LHS) of a Datalog clause is called its **head** and the right hand side (RHS) is called its **body**. The body of a clause can be empty. Clauses with an empty body represent *facts* (EDB relations), and clauses with at least one literal in the body represent *rules* (IDB relations).

Any Datalog program P must satisfy the following **safety conditions**[BaRa86] :

(1) Each fact of P is *ground*. (i.e. it does not contain any variables.)

(2) Each variable which occurs in the head of a rule must also occurs in the body of the same rule.

These conditions guarantee that the set of all facts that can be derived from a Datalog program is finite.

From the syntactical point of view, Datalog is a subset of Prolog, thus each set of Datalog clause could be parsed and executed by a Prolog interpreter. However, *Datalog and Prolog differ in their semantics.*

Datalog has been developed for applications which use a large number of facts stored in a relational database. Therefore, we always consider two disjoint sets of clauses, i.e. a set of ground facts: the Extensional Database (EDB), and a Datalog program P: the Intensional Database (IDB).

Ground facts are stored in a relational database. We assume that each EDB predicate r corresponds to exactly one relation R of our database, such that each fact $r(c_1, ..., c_k)$ of the EDB is stored as a tuple $< c_1, ..., c_k >$ of R.

The IDB predicates of P can be identified with relations, called the IDB relations, or derived relations. IDB relations are not stored explicitly and they correspond to the concept of "view" in relational databases which is calculated upon querying. IDB relations can be defined using recursive rules. "The materialization of these *recursive views*, i.e. their effective and efficient computation is the main task of a Datalog compiler or interpreter." [CeGoLe89]

In summary, because Datalog is a database language, it has significant semantic differences from Prolog[CeGoLe89]:

1. Datalog has a purely declarative semantics while Prolog is defined by operational semantics.
2. Datalog uses a set-oriented data processing strategy while Prolog returns query results in a one-tuple-at-a-time fashion.

A Prolog program proceeds according to a resolution strategy which uses a *depth-first search method with backtracking* for constructing proof trees and respect the order of the clauses and literals as they appear in the program [StSh86]. This strategy does not guarantee termination. The termination of a recursive Prolog program depends strongly on the order of the rules in the program, and on the order of the literals in the rules.

Example 1-2. The difference between Prolog and Datalog.
Suppose we rewrite the same generation program in Example1-1 as P':

$sg(X,Y)$:- $sg(X_1, Y_1)$, $parent(X, X_1)$, $parent(Y, Y_1)$. (r_1')
$sg(X, X)$:- $person(X)$. (r_2')

This program P' differs from program P of Example1-1 only by the order of the rules and by the order of literals in the rule bodies. From a Datalog point of view, the order of clauses and literals is totally irrelevant, thus P and P' are equivalent. On the other hand, suppose we use a Prolog interpreter to evaluate the same query "?- sg(Anne, X)", P and P' will be evaluated differently. If P is used, Anne's same generation cousins will be returned one by one. But if P' is used, we would run into an infinite recursion without getting any result. □

In spite of this, efforts have been made to couple Prolog to an external database[CeGoLe89]. A Prolog interpreter can be designed to distinguish between IDB and EDB predicates. When an EDB goal is encountered during the execution of a Prolog program, the interpreter tries to retrieve a matching tuple from mass memory. Due to the procedural semantics of Prolog, which prescribes a particular order of visiting goals and sub-goals, the required interaction between the interpreter and the external database is of the type one tuple at a time. This method of accessing mass memory is quite inefficient compared to the set-oriented methods used by high-level query languages such as SQL.

Because of these reasons, although Prolog is a rich and flexible logic programming language which has gained enormous popularity over the last decade in many application domains, it is not an efficient and practical database language. One goal of Deductive Database query languages, such as Datalog, is to provide flexible and efficient access to large quantities of data stored in mass memory.

1.4. Thesis Overview

So far, we have generally discussed the major concepts of Deductive Database and its fundamental query language: Datalog. We will now concentrate on one of the most essential issues of Deductive Database research: *recursive query processing*.

The focus of this thesis is on the implementation aspects of recursive query processing, especially on a recursion optimization method : *the chain-based compilation and evaluation method on linear recursions*. For the details on Deductive Database formalisms, please refer to the literature references[GrMi92, Ullm89a, BrJa84] listed at the end of this thesis. However, a survey of some of the most well-known evaluation and optimization techniques on recursive query processing using deductive database techniques is presented in Chapter 2.

Starting at Chapter 3, the discussion focuses on the chain-based compilation and evaluation method. Chapter 3 concentrates on the compilation part, i.e. *the automatic generation of compiled chain forms for function-free single linear recursions* and its implementation considerations. Chapter 4 discusses the analysis/evaluation part and the extension of this method to the domain of *nested functional linear recursions*. A typical case, the n-queens program will be used to illustrate how this method works and compare it with the conventional Prolog approach.

Chapter 5 concludes the thesis by comparing the chain-based evaluation method with other recursion processing methods and discussing its possible further extensions.

Chapter2
Deductive Database and Its Implementation Techniques

In this chapter, we discuss the area of *recursive query processing* in Deductive Databases and present a survey on the most well-known techniques in this area.

2.1 A Deductive Database Model: Logic and Databases

First.of all, we summarize the deductive database terminologies and some special assumptions our discussion will be based on.

(1) The Two Components of a DDB: EDB and IDB

A Deductive Database consists of an **extensional database(EDB)**, an **intensional database(IDB)** and a set of **integrity constraints** (ICs). The EDB is a large set of ground facts stored in a relational database. The IDB is a set of intensional predicates defined by a set of function-free Horn clauses called deduction rules. The ICs are not included in our discussion here.

For the ease of analysis, we assume that each predicate symbol either denotes an EDB relation or an IDB relation, but not both. In another word, we decompose the database into a set of pure base predicates and a set of pure derived predicates. It is easy to prove that such a decomposition is always possible[Ullm89a].

(2) The Syntax of Datalog

Datalog is a typical data definition language for Deductive Databases.

According to the syntax of Datalog, both facts and rules are represented as Horn clauses in the form of "L_0 :- L_1, ..., $L_{n.}$" where each L_i is a literal of the form $p_i(t_1, ...t_{ki})$ such that p is a predicate of arity n and t_j is a constant or a variable.

The left hand side of a Datalog clause is called its **head** and the right hand side is called its **body**. The body of a clause may be empty. Clauses with an empty body represent facts while clauses with at least one literal in the body represent rules.

As a notational convention, we use strings starting with upper case letters to denote variables and strings starting with lower case letters to denote relations (or predicates) and constants. For a given Datalog program, it is always clear from the context whether a particular nonvariable symbol is a constant or a predicate symbol.

For simplicity, we require that all literals with the same predicate symbol are of the same arity, i.e. they have the same number of arguments.

A literal which does not contain any variables is called an **instantiated literal**.

Any Datalog program P must satisfy the following safety conditions[BaRa86]:

- Each fact of P is ground, i.e. it does not contain any variables.
- Each variable which occurs in the head of a rule also occur in the body of the same rule.

These conditions guarantee that the set of all facts that can be derived from a Datalog program is finite.

(3) The Logical Semantics of Datalog

Considering Datalog as a simplified version of Logic Programming, its semantics can be easily described in terms of model theory.

In this point of view, we see rules as defining possible worlds or **models**. An interpretation of a collection of predicates assigns truth or falsehood to every possible instance of those predicates, where the predicate's arguments are chosen from some infinite domain of constants. Usually an **interpretation** is represented by its set of true instances. To be a model of a set of rules, an interpretation must make the rules true, no matter what assignment of values from the domain is made for the variables in each rule.

Example 2-1. Consider the following Datalog program[Ullm89a]:

```
p(X) :- q(X).        (r1)
q(X) :- r(X).        (r2)
```

Suppose the domain of the variable X is integer.

Intuitively, these rules state the fact that " If r is true for a certain integer, then q is also true; and whenever q is true, p is true as well. "

Let's assume that in this case, r is an EDB predicate while p and q are IDB predicates defined in terms of r. We also assume that r(X) = true iff X=1.

The model $M_1 = \{ r(1), q(1), p(1), q(2), p(2), p(3) \}$ is a possible interpretation of the database because it satisfies all the rules and facts defined in this program.

However, there is another consistent model $M_2 = \{ r(1), p(1), q(1) \}$ which also satisfies the program. In fact, there are an infinite number of models consistent with the EDB that has only r(1) true. But M_2 is special because it is a minimal model, i.e. we can not make any true fact false and still have a model consistent with the database { r(1) }. □

Formally, given a set of relations for the EDB predicates, say $R_1, ..., R_k$, a **fixed point** of the Datalog equations (with respect to $R_1, ..., R_k$) is a solution for the relations corresponding to the IDB predicates of these equations.

It has been proved that each Datalog program has a unique minimal model containing any given EDB relations, and this model is also the unique minimal fixed point with respect to those IDB relations. A solution or fixed point S_0 is **the least fixed point** with respect to a set of EDB relations $R_1, ..., R_k$ if for any solution S, we have $S_0 \subseteq S$ ([Ullm89a]).

We always use the least fixed point as the model of the database. (Notice that if we allow evaluable arithmetic predicates, the minimal model is generally infinite.)

(4) The Query and the Adornment of a Predicate

We use "adornments" to represent the instantiation information of a predicate. Let $p(X_1, X_2, ..., X_n)$ be an n-ary predicate. The **adornment** of p is a sequence of length n of b's and f's $(a_1...a_n)$ where a_i is 'b' if the i-th varible of p is bounded or instantiated, otherwise it is 'f'. Adornments are denoted as superscripts. For example, parent(adam, X) is denoted as $parent^{bf}(adam, X)$.

A **query** is an adorned predicate with an instantiation of the bounded variables. For example, ?-parent^{bf}(adam,X) is a query.

Having discussed the basic terminologies, we now get to the point of discussing **recursions**.

2.2 A Classification of Recursions

Definition.

A predicate p is said to **imply** a predicate r ($p \rightarrow r$) if there is a Horn clause in IDB with predicate r as the head and p in the body, or there is a predicate q where $p \rightarrow q$ and $q \rightarrow r$ (transitivity). A **predicate r is recursive** if $r \Rightarrow r$. ($\Rightarrow$ is defined is the transitive closure of $\rightarrow$)

Two predicates p and q are **mutually recursive** if $p \Rightarrow q$ and $q \Rightarrow p$. Mutual recursion is an equivalence relation on the set of recursive predicates.

A rule p :- p_1, p_2, ..., p_n is recursive iff there exists p_i in the body of the rule which is mutually recursive to p. A recursive rule is **linearly recursive** if there is one and only one predicate p_i in the body of the rule which is mutually recursive to p.

A **rule cluster** of predicate r is the maximal subset of rules in IDB in which all the head predicates of the rules are either r or p where $p \Rightarrow r$. If r is recursive, the r-cluster is called the **recursive-cluster**. A recursive cluster is linear, or more precisely, **single linear** if it consists of one linear recursive rule and one or more nonrecursive rules (**exit rules**).

Example 2-2. The predicate sg (same_generation) is defined as follows:

sg(X,X) :- person(X). (r_1)

sg(X,Y) :- parent(X,X_1), sg(X_1, Y_1), parent(Y,Y_1). (r_2)

According to the above classification, sg is a recursive predicate; r_2 is a linearly recursive rule and the rule cluster of sg is a single linear recursive rule cluster where r_1 is the exit rule and r_2 is the only recursive rule. □

Although recursions can be much more complicated than linear recursions, we find that a large set of database-oriented applications belong to this relatively simple domain. Thus, many techniques studied in this thesis are confined to linear recursions. Nevertheless, we do find some interesting applications which involve recursions with function symbols (such as arithmetic functions and list processing functions) and multiple "levels" of linear recursions. We will discuss how to handle these extensions to linear recursions as well.

Here are some examples of the various kinds of recursions :

Example 2-3. The ancestor relation:

ancestor(X,Y) :- parent(X,Y). (r_1)

ancestor(X,Y) :- ancestor(X, Z), parent(Z,Y). (r_2)

Ancestor is a recursive predicate and its rule cluster is single linear. □

Example 2-4. A random linear recursion

r(X,Y, Z) :- a(X,Y), r(X_1, Z, Z_1), b(X_1, Z_1). (r_1)

r(X,Y, Z) :- e(X,Y,Z). (r_2)

In this example, r is recursive and the cluster is single linear recursive. We will be looking into this example in Chapter 3 because it is a typical case that the Magic Sets method (which is one of the most popular optimization methods on recursive query optimization in Deductive Databases) has some inherent problems in binding propagation[Han91b]. □

Example 2-5. A functional linear recursion.

append([], L, L). (r_1)

append([X|L_1], L_2, [X|L_3]) :- append(L_1, L_2, L_3). (r_2)

This is a single linear recursion with list function symbols. We will discuss its implementation in Chapter 4. □

Example 2-6. An airline reservation example [Han91a]:

travel ([Fno], Dep, DTime, Arr, ATime, Fare)

:- flight (Fno, dep, DTime, Arr, ATime, Fare). (r_1)

travel ([Fno|L], dep, DTime, Arr, ATime, Fare)

:- flight(Fno, dep, DTime, Int, IATime, F_1),

travel(L, Int, IDTime, Arr, ATime, S_1),

Fare = F_1 + S_1. (r_2)

This is a functional linear recursion with both list functions and arithematic functions. □

Example 2-7. The n-queens problem:

```
nqueens(N, Qs) :-  range(1, N, Ns), queens(Ns, [], Qs).

range(M, N, [M|Ns]) :- M<N, M1 is M+1, range(M1, N, Ns).
range(N, N, [N]).

queens( Unplaced, safe, Qs) :- Select(Q, Unplaced, Unplaced1),
                               not attack(Q, Safe),
                               queens(Unplaced1, [Q|Safe], Qs).
queens( [], Qs, Qs).

attack(X, Xs) :- attack1(X, 1, Xs).

attack1(X, N, [Y|Ys]) :-  X is Y+N.
attack1(X, N, [Y|Ys]) :- X is Y-N.
attack1(X, N, [Y|Ys]) :- N1 is N+1, attack1(X, N1, Ys).

select(X, [X|Xs], Xs).
select(X, [Y|Ys], [Y|Zs]) :- select(X, Ys, Zs).
```

This recursion consists of multiple levels of linear recursions, which is more formally named as **nested linear recursion**. We will discuss how to process queries against "nqueens" in a deductive database approach in Chapter 4. This example is interesting because it is a good showcase on how the DDB approach is different from that of Prolog, and what the advantages/disadvantages are of each. However, for this particular example, there are no associated EDB relations. □

2.3 Characteristics of the Strategies

Before discussing the evaluation methods, it would be helpful to see some of the criteria used to distinguish them. We will briefly look at three such criteria [CeGoLe89]:

- Bottom-up vs. Top-down strategy
- Query evaluation vs. Query optimization
- Syntactic optimization vs. Semantic optimization

2.3.1 Search Strategy: Bottom up vs. Top Down

The evaluation of a Datalog goal can be performed in two different ways:

- **bottom-up**: starting from the existing facts and inferencing new facts
- **top-down**: starting from the query, trying to verify the premises which are needed in order for the conclusion to hold.

We consider these two evaluation strategies as representing different interpretations of a rule[CeGoLe89].

The Bottom-up evaluation methods consider rules as *productions* that apply the initial program to the EDB and produce all the possible consequences of the program, until no new facts can be deduced. The bottom-up method can naturally be applied in set-oriented fashion, i.e. taking as input the entire relations of EDB. This is a desirable feature in the deductive database context because large amount of data must be retrieved from mass memory. On the other hand, the bottom-up methods do not take immediate advantage of the selectivity due to the existence of arguments bound to constants in the goal predicate. The selection is usually done on the final result set.

Most of the methods we will be discussing belong to this category, such as Naive/Semi-Naive method [2.4.1], the Magic Sets method [2.4.3], the Counting/Reverse Counting method [2.4.4] and the Chain-Based Compilation and Evaluation method [2.4.5]. To take advantage of query bindings and other available constraints, three of these methods are optimization methods [2.4.3,2.4.4,2.4.5] which do some pre-processing (compilation or rule-rewriting) on the original rule set before actually retrieving data from EDB to eliminate irrelevant data at an earlier stage.

On the other hand, top-down evaluation treats rules as *problem generators*. Each goal is considered as a problem that must be solved. The initial goal is matched with the left-hand side of some rules and generates other problems corresponding to the right-hand side predicates of that rule. This process continues until no new problems are generated.

In this case, if the goal contains some bounded argument, then only the facts that are relevant to the goal constants are involved in the computation. Thus, this evaluation method performs an optimization because the computation automatically disregards many of the facts which are not useful for producing the results. However, using top-down methods, it is more natural to produce the answers in one-tuple-at-a-time fashion which is not desirable in database applications.

Furthermore, we can distinguish two search strategies in top-down approach, i.e. **breadth-first** or **depth-first**. With the depth-first approach, we face the disadvantage that the order of literals in rule bodies strongly affects the performance (especially termination) of methods. This happens in Prolog, where not only efficiency but also the termination of programs is affected by the left-to-right order of subgoals in the rule bodies[StSh86]. Instead, Datalog goals are more natually executed through breadth-first technologies, thus the result of computation is neither affected by the order of predicates in the body of a rule, nor by the order of the rules in a program. The Query-Subquery method [2.4.2] applies top-down searching strategy.

2.3.2 Objective : Query Evaluation vs. Query Optimization

The second classification criterion is based on the different objectives of the methods [BaRa86]:

Query evaluation methods: These strategies consist of an actual evaluation algorithm, i.e. a program which, given a query and a database, will produce the answers to the query. The query evaluation methods that we will discuss are Naive/Semi-Naive evaluation and Query-Subquery approach .

Query optimization methods: Strategies in this class assume an underlying simple evaluation strategy and optimize the rules to make their evaluation more efficient, based on the current IDB definitions and sometimes the particular query. These methods are usually referred to as "rewriting methods". We will discuss the Magic Sets method, Counting/Reverse-Counting method and the Chain-Based Evaluation method.

2.3.3 Type of Optimization: Syntactic Optimization or Semantic Optimization

Optimization methods differ in the type of information used [CeGoLe89] .

Syntactic optimization is the most commonly used. It deals with those transformations to a program which are based on the program's *syntactic features*. We usually consider two kinds of structural properties: one is *the program structure*, i.e. the types of the rules which constitute the program. For example, the methods we will be discussing are mostly based on the *linearity* of the recursive rules to produce optimized form of evaluation. The other one is *the structure of the goal or query*. Particularly, we consider the selectivity or *query bindings* that comes from goal constants. These two approaches are not mutually exclusive. Actually most optimization methods combine both of them.

Semantic Optimization, on the other hand, concerns the use of additional semantic knowledge about the database in order to produce an efficient answer to a query. More importantly, the combination of the query with additional semantic information is performed *automatically*. Semantic methods are often based on *integrity constraints*, which express properties of valid databases. For example, in example2-6, we have the constraint that the arrival time of a flight is always later than its departure time, thus, when we construct a chain of consecutive flights, only the flights with a departure time later than the arrival time of the previous flight should be considered.

Although semantic optimization has the potential of significant improvements on query processing strategies, it is more case-specific. We are not going to discuss semantic optimization methods here. Interested reader can find references listed in [CeGoLe89] and [Han91a].

Lastly, there are three important properties all recursion processing methods must satisfy[CeGoLe89]:

1. The method must be **sound**: it should not include in the result set tuples which do not belong to it.
2. The method must be **complete**: it must produce all the tuples of the result.
3. The method must be **terminable**: the computation must be performed in finite time.

Although we are not going to provide formal proofs, the methods studied in the next section satisfy all these properties.

2.4 Survey of Evaluation Methods and Optimization Techniques

We will use the example of same-generation to illustrate how the methods work.

Example 2-8. A sample DDB:

The Intensional Database is :

sg(X, X) :- person(X). (r_1)

sg(X,Y) :- parent(X, X_1), sg(X_1, Y_1), parent(Y, Y_1). (r_2)

The Extensional Database is:

person: { (a), (b), (c), (d), (e), (g), (h) }

parent: { (d, g), (e, g), (b, d), (a, d), (a, h), (c, e) }

The sample query is: "?- sg(a, X). "

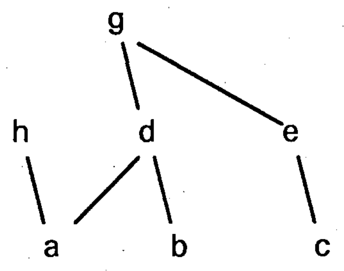


Figure 2-1. The Family Tree

□

2.4.1. Naive and Semi-Naive Evaluation Methods

Naive evaluation is a bottom-up, iterative evaluation strategy. The method is also well-known in numerical analysis as Gauss-Seidel method, used for determining the iterative solution (fixpoint) of system of equations.

Assume the following relational equation: $R_i = Eval_i(R_1, ..., R_n)$,$(i = 1, ..., n)$. The Naive Evaluation Method works as follows [BaRa86] :

Initially, the relations R_i $(i = 1, ..., n)$ are set to the empty set. Then, the computation $R_i := Eval_i(R_1, ..., R_n)$ $(i = 1, ..., n)$ is iterated until all the R_i's do not change between two consecutive iterations (i.e. until R_i's have reached its *fixedpoint*). At the end of the computation, the value assumed by the variable relations R_i is the solution of the system.

Algorithm2-1. The Naive Evaluation Method [CeGoLe89]

Input: A system of IDB relations $R_1, ..., R_n$ and an EDB.

Output: The least fixed point of the IDB relations $R_1, ..., R_n$.

Method:

```
For i:= 1 to n do $R_i := \emptyset$;
    Repeat
        nochange := true;
        For i:= 1 to n do
          begin
              S := $R_i$;   $R_i := Eval_i(R_1, ..., R_n)$;
              if $R_i \neq S$ then nochange := false;
          end
    until nochange;
For i :=1 to n do output($R_i$). □
```

For example2-8, to compute the sg relation using the Naive Evaluation method, we have in each iteration:

$sg_1(X,Y)$ = { (a,a), (b,b), (c,c), (d,d), (e,e), (g,g), (h,h) }.

$sg_2(X,Y)$ = { (a,a), (b,b), (c,c), (d,d), (e,e), (g,g), (h,h), (d,e), (e,d), (a,b), (b,a) }.

$sg_3(X,Y)$ = { (a,a), (b,b), (c,c), (d,d), (e,e), (g,g), (h,h), (d,e), (e,d), (a,b), (b,a), (a,c), (b,c), (c,a), (c,b)}.

$sg_4(X,Y)$ = { (a,a), (b,b), (c,c), (d,d), (e,e), (g,g), (h,h), (d,e), (e,d), (a,b), (b,a), (a,c), (b,c), (c,a), (c,b)}.

Since $sg_3 \equiv sg_4$, for the query ?- sg(a, Y), we have the result set Y= {a, b, c}. □

Notice that the Naive Evaluation method is a **bottom-up evaluation method**. It proceeds in a set-oriented way, and the order of rules and the order of predicates in the body of a rule do not affect the algorithm.

The Naive Evaluation is probably the simplest and the most natural way of processing recursive queries in a set-oriented manner. However, we can easily see two weaknesses of this method: (1) it involves a lot of duplication of calculation at each iteration; (2) it does not take advantage of query instantiation information to make the algorithm work more efficiently.

To deal with the first problem, the **Semi-Naive Evaluation Method** is introduced. Basically, it takes the same approach as Naive Evaluation method, except that it is designed to eliminate some redundancies in the evaluation of tuples at different iterations[Ullm89a] .

In the particular case of *linear recursions*, the Semi-Naive method proceeds by evaluating only the new tuples generated at each iteration and terminates when no more new tuples are generated [BaRa86].

2.4.2 Query/Subquery Method

The Query/Subquery (QSQ) Method is a *top-down evaluation method* based on backward chaining. The advantage of this method is that, it takes into consideration the query instantiation information at the very first stage, and tries to access the minimum number of facts in order to determine the answer.

The key concept of this method is *subquery*. A goal, together with the program, determines a **query**. Literals in the body of any one of the rules defining the goal predicate are subgoals of the given goal. A subgoal, together with the program, defines a **subquery**. In order to answer the query, each goal is expanded in a list of subgoals, which are then expanded in their turn.

The method memorizes two sets at each stage of computation: (i) a set Q of *generalized subqueries*, which contains all the subgoals that are currently under consideration, and (ii) a set R of *derived relations*, containing answers to the main goal and answers to intermediate subqueries.

The QSQ algorithm is basically doing two things: generating new answers to the R set and generating new subqueries that must be answered to the Q set.

There are two versions of the Query/Subquery algorithm: **Iterative Query/Subquery (QSQI)** and **Recursive Query/Subquery (QSQR)**. The difference between these two versions is which set (Q or R) is considered first[CeGoLe89]. QSQI privileges the production of answers, i.e. when a new query is encountered, it is suspended until the end of the production of all the possible answers that do not require using the new subquery. QSQR, on the other hand, behaves in the other way around: whenever a new query is found, it is recursively expanded and the answering to the current subquery is postponed until the new subquery has been completely solved.

At the end of the computation, R contains the answers to the goal. As in Naive/Semi-Naive method, a final selection is needed to generate the answers. However, the query instantiation information has been considered in the initial goal, thus the size of the relations involved in the computation is comparatively much smaller than those involved in Naive evaluation.

The two algorithms are described as follows:

Algorithm2-2. Iterative Query/Subquery Algorithm **QSQI** [BaRa86] :

```
Initial state is <{query(X)}, {}>
while the state changes do
  for all generalized queries in Q do
    for all rules whose head matches the generalized query do
        begin
        (i) unify rule with the generalized query;
            ( i.e. propagate the constraints. This generates new generalized queries
              for each derived predicate in the body by looking up the base relations. )
        (ii) generate new tuples;
            ( by replacing each base predicate on the right by its value and every derived
              predicate by its current temporary value.)
        (iii) add these new tuples to R;
        (iv) add these new generalized queries to Q;
        end. □
```

Algorithm2-3. Recursive Query/Subquery Algorithm **QSQR**[BaRa86]:

```
Initial state is < {query(X)}, {} >
evaluate(query(X)).
procedure evaluate( q )  /* q is a generalized query */
begin
  while "new tuples are generated" do
     for all rules whose head matches the generalized query do
     begin
        unify the rule with the generalized query; (i.e. propagate the constants)
        until there are no more derived predicates on the right do
        begin
        (i) choose the first/next derived predicate according to the selection function;
        (ii) generate the corresponding generalized query;
           (This is done by replacing in the rule each base predicate by its value and each
             previously solved derived predicate by its current value).
        (iii) eliminate from that generalized query the queries that are already in Q;
        (iv) this produces a new generalized query q', add q' to Q;
        (v)  evaluate (q')
        end
      replace each evaluated predicate by its value and evaluate the generalized query q;
      add the results in R;
      return the results;
      end
end □
```

This method can be compared with Prolog because they both use the top-down strategy. The differences are : (1) Prolog proceeds one tuple at a time while QSQ is set-oriented, because it processes the whole relation (generalized queries) at each step. In this sense, QSQ is more appropriate for database applications. (2) QSQ adopts breadth-first search and always terminates, while Prolog applies depth-first search and may not terminate in some cases.

The query/subquery algorithm was first introduced by L.Vieille [CeGoLe89]. A compiled version of QSQR has been implemented on top of the INGRES relational system [BaRa86].

2.4.3. The Magic Sets Method

The method of **Magic Sets** is a logical rewriting method (optimization method) that generates from the given set of rules a new (and larger) set of rules, which are equivalent to the original set with respect to the query, and its bottom-up evaluation is more efficient.

The idea of the Magic Set method is to use query binding information to cut down on the number of potentially relevant facts involved in the computation. The new rules involved in the rewriting part act as *constraints*, which force the program variables to satisfy some additional conditions, based on the information of query bindings.

We use example 2-8 to show how it works, more details can be found in [BaRa86, BMSU86].

The original rule set is :

sg(X,X) :- person(X).

sg(X,Y) :- parent(X, X_1), parent(Y, Y_1), sg(X_1, Y_1).

The query is: "query(X) :- sg(a, X)." i.e. "find person a's same generation cousins".

The following steps are taken to generate a "Magic Set" from the original rules based on the provided query constraints (bindings) [BaRa86] :

Step 1. Generate Adorned Rules

For each rule r and for each adornment a of the predicate on the left, generate an adorned rule: (i) Define recursively an argument of a predicate in the rule r to be *bounded* if either it is bounded in the adornment a, or it is a constant, or it appears in a base predicate that has a bounded variable. These bindings are propagated through the base predicates. (ii) The adorned rule is obtained by replacing each derived literal by its adorned version.

For example, the rule

$sg(X,Y)$:- $p(X, X_1)$, $p(Y, Y_1)$, $sg(X_1, Y_1)$.

with the adornment "bf" on the head predicate (which means X is bounded and Y is free) would generate the adorned rule : (Note that we assume all the non-recursive predicates are base relations and do not have functions involved. Thus, only the adornment of IDB predicates needs to be considered.)

$sg^{bf}(X, Y)$:- $p(X, X_1)$, $p(Y, Y_1)$, $sg^{bf}(X_1, Y_1)$.

Note that for a set of R rules with the same head predicate, the adorned system is of the size K*R, where K is a factor exponential to the number of attributes per derived predicate. Among these adorned rules, we only consider the ones that derive the query.

In our example, the reachable adorned system is:

$sg^{bf}(X,Y)$:- $p(X, X_1)$, $p(Y, Y_1)$, $sg^{bf}(X_1, Y_1)$.

$sg^{bf}(X, X)$:- person(X).

$query^{f}(X)$:- $sg^{bf}(a, X)$.

This new set of rules is equivalent to the original set *in respect to the query* $sg^{bf}(a,X)$.

Step 2. Generate Magic Rules

For each occurrence of a derived predicate on the right of an adorned rule, we generate a magic rule [BaRa86] :

(i) Choose an adorned literal p on the right of an adorned rule r.

(ii) Erase all the other derived literals on the right.

(iii) In the derived predicate occurrence, replace the name of the predicate by magic.p^a where a is the literal adornment and erase the non-distinguished variables.

(iv) Erase all the non-distinguished base predicates.

(v) On the left hand side, erase all the non-distinguished variables and replace the name of the predicate by magic.$p_1^{a'}$, where p_1 is the predicate on the left and a' is its adornment.

(vi) Exchange the two magic predicates.

For example,

$sg^{bf}(X, Y)$:- $p(X, X_1)$, $p(Y, Y_1)$, $sg^{bf}(X_1, Y_1)$.

generates the magic rule

$magic^{bf}(X_1)$:- $p(X, X_1)$, $magic^{bf}(X)$.

Notice that the magic rules simulate the bound of arguments through backward chaining.

Step 3. Generate Modified Rules.

For each adorned rule generated in Step1, we generate a modified rule : for each rule whose head is p.a , add on the right hand side the predicate magic.p.a(X), where X is the list of distinguished variables in that occurrence of p.

For instance, the adorned rule:

$sg^{bf}(X,Y)$:- $p(X, X_1)$, $p(Y, Y_1)$, $sg^{bf}(X_1, Y_1)$.

generates the modified rule:

$sg^{bf}(X, Y)$:- $p(X, X_1)$, $p(Y, Y_1)$, $magic^{bf}(X_1)$, $sg^{bf}(X_1, Y_1)$.

Combining Steps 1, 2 and 3, the complete modified set of rules is as follows:

sg(X, X) :- person(X).	(r_1')
sg(X, Y) :- magic(X_1), p(X, X_1), p(Y, Y_1), sg(X_1, Y_1).	(r_2')
magic(a).	(r_3')
magic(X_1) :- magic(X), p(X, X_1).	(r_4')

After the rule rewriting, the IDB relation "magic" contains all the ancestors of a. The tuples of the relation "magic" defined by the magic rules form **the magic set**. By computing the magic set, we impose the restriction that, to compute a's same generation cousins, we only need to consider those pairs of same generations whose first element is an ancestor of a. This is essentially the whole purpose of the rule rewriting: to *cut down on irrelevant facts at an earlier stage of computation*.

After the magic sets transformation, the resulting program can be evaluated by a simple bottom-up algorithm, such as Naive or Semi-Naive method, but still takes advantage of the binding information.

The original idea of Magic Set method was presented in [BMSU86]. The method has been extended by Sacca and Zaniolo to a class of queries to logic programs that contain function symbols [CeGoLe89]. So far, the Magic Sets method is one of the most popular methods used in DDB implementations.

2.4.4. The Counting Method

The **counting method** is derived from the Magic Set method. It applies under two conditions: (i) the data is acyclic and (ii) the program is linear and has at most one recursive rule for each predicate [BMSU86] .

Consider again the same_generation example. The Magic Set method restricts the computation to the ancestors of person "a". Furthermore, the counting method maintains the information of the person's "distance" to person "a" in the generation tree, i.e. whether it is one of a's parents(distance = 1), grandparents(distance = 2), grand-grand parents (distance = 3), etc. Thus further computation is restricted to the children of a's parents, grandchildren of a's grandparents, and grand-grand children of a's grand-grand parents, etc.

Here is the result rule set of applying counting transformation to the output of Magic Sets method on the query ?- sg(a, X).

> sg(X, X, I) :- person(X), integer(I).
>
> sg(X, Y, I) :- parent(X, X_1), sg(X_1, Y_1, J), parent(Y, Y_1), counting(X_1, I), I=J-1.
>
> counting(a, 0).
>
> counting(X_1, I) :- counting(X, J), parent(X, X_1), I= J+1.

Generally, if we have a "general" single linear rule system:

> r(X, Y) :- flat(X, Y).
>
> r(X, Y) :- up(X, XU), r(XU, YU), down(YU, Y).
>
> query(X) :- r(a, X).

The resulting rule set after applying counting method is:

```
counting(a, 0).
counting(X, I) :- up(Y, X), counting(Y, J), I=J+1.
r'(X, Y, I) :- counting(X, I), flat(X, Y).
r'(X, Y, I) :- counting(X, I), up(X, XU), r'(XU, YU, J), down(YU, Y), I=J-1.
query(X) :- r'(a, X, 0).
```

Obviously, if the base relations are cyclic, the counting method encounters the termination problem. Also, special considerations should be put with the evaluation of built-in functions, such as "+" and "-". We will discuss the evaluation of function symbols in more detail later, but generally the rule is to defer the computation until enough binding information has been gathered to generate a finite result set.

The original idea of the counting method was introduced in [BMSU86], a similar method called "reversed counting" is discussed in the same paper.

2.4.5 The Chain-Based Compilation and Evaluation Method

To conclude the survey, we discuss the general idea of the chain-based compilation and evaluation method. The implementation and extension of this method is the focus of this thesis.

Chain-based compilation method is a *query-independent* compilation method. The idea is that, based on the *expansion regularity* of recursive rules, a single linear recursive rule cluster can be compiled into a highly regular **chain-form** (or **linear normal form**) which captures query bindings that are difficult to be captured in other methods and facilitates efficient query analysis and evaluation [Han89a] .

We define the **first expansion** of a linear recursive rule as itself, and the **k-th expansion** of a linear recursive rule as the unification of the (k-1)th expansion and the definition recursive rule.

Let's look at the expansion behavior of the recursive rule for sg:

The first expansion (the definition rule) is :

$sg(X, Y)$:- $parent(X, X_1), sg(X_1, Y_1), parent(Y, Y_1)$.

The second expansion is:

$sg(X, Y)$:- $parent(X, X_1), parent(X_1, X_2), sg(X_2, Y_2), parent(Y_1, Y_2), parent(Y, Y_1)$.

The third expansion is:

$sg(X, Y)$:- $parent(X, X_1), parent(X_1, X_2), parent(X_2, X_3)\ sg(X_3, Y_3),$
$parent(Y_2, Y_3), parent(Y_1, Y_2), parent(Y, Y_1)$.

and so on.

We can see that the variable connection patterns in these expansions is highly regular and the variables in the head predicate and the variables in the recursive predicate in the body are connected by a "chain" of predicates.

If we describe such a **chain** as:

$$parent^i(X_{i-1}, X_i) = \begin{cases} True & i = 0 \\ parent(X_{i-1}, X_i) & i = 1 \\ parent^{i-1}(X_{i-1}, Z), parent(Z, X_i) & i > 1 \end{cases}$$

The i-th ($i > 0$) expansion of sg is:

$sg_{(i)}(X, Y) = parent^i(X, X_i), sg(X_i, Y_i), parent^i(Y, Y_i)$.

Since the definition of "sg" is the union of all its expansions, we have the **chain-form** of "sg" as:

$$sg(X, Y) \ = \ \bigcup_{i=0}^{\infty} (\ parent^i(X, X_i),\ parent^i(Y, Y_i),\ (person(X_i),\ X_i{=}Y_i)\ ).$$

Thus, sg is a "double-chain recursion", with two chains: $parent^i(X, X_i)$ and $parent^i(Y,Y_i)$. These two chains are actually two binding propagation paths.

Notice that:

- The compilation process is independent of any query forms. (Thus the name "query independent compilation".)
- The chain form depends only on the logical definition of the rule. The order of rules specified in the rule cluster and the order of literals specified in the body of rules are not relevant to the result of compilation.
- It has been proved that all single linear recursions can be compiled into either bounded recursion or regulated chain forms [Han89a] .

Each chain in the compiled form has two ends: **the near end** of the chain shares variables with the exit expression while **the far end** is the other end which connects to queries. For example, considering the chain $parent^i(X, X_i)$ in the case of "sg", X is the far end and X_i is the near end.

Based on the chain form, we can generate efficient evaluation plans for various queries. The basic considerations in the evaluation phase are:

1. Choose the more selective end of the chain to start processing.
2. Based on the query requirements, choose an evaluation strategy between query closure or existence checking [Han89b] .

More complicated situations, such as functional recursions and multi-level recursions require more considerations such as chain-splitting[HaWa91], which will be discussed in Chapter 4.

Considering again the query "?- sg(a, Y)", we can see that, for the chain $\text{parent}^i(X, X_i)$, the far end is bounded (X = a) while the near end is free. Thus, the query process should start from the far end of this chain, then the exit part, and proceed "downwards" along the other chain.

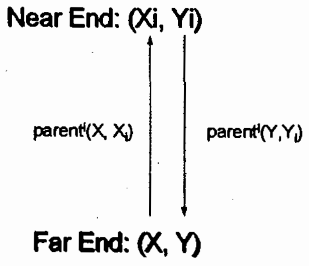


Figure 2-2. The Chain Following Directions

The evaluation proceeds as follows:

i=0, $sg_{(0)}(a, Y) =$ person(a), a=Y.

which derives Y = {a}.

i=1, $sg_{(1)}(a, Y) =$ parent(a, X_1), (person(X_1, Y_1), $X_1 = Y_1$), parent(Y, Y_1).

which derives in the following order: X_1 = {d, h}; Y_1 = {d, h}; Y={a,b}.

i=2, $sg_{(2)}(a, Y) =$ parent(a, X_1), parent(X_1, X_2), (person(X_2, Y_2), X_2=Y_2),

parent(Y_1, Y_2), parent(Y, Y_1).

which derives the following result set: X_1={d, h}; X_2={g}; Y_1={d,e}, Y={a,b,c}.

i=3, $sg_{(3)}(a, Y) =$ parent(a, X_1), parent(X_1, X_2), parent(X_2, X_3),

(person(X_3, Y_3), X_3=Y_3),

parent(Y_2, Y_3), parent(Y_1, Y_2), parent(Y, Y_1).

which derives: X_1 = {d,h}, X_2 = {g}, $X_3 = \varnothing$. Processing terminates.

Thus, we have the result set Y = {a} ∪ {a,b} ∪ {a,b,c} = {a, b, c}. □

The key to this method is to generate the compiled chain form. The idea is proposed in [Han89a]. In the next chapter, we will discuss in detail how to automatically generate the chain form. In Chapter 4 we will discuss the query analysis and evaluation based on chain forms.

Chapter 3
Compilation of Linear Recursions

3.1. Difficulties of Magic Rule Rewriting

As we discussed in Chapter 2, many techniques have been developed to evaluate recursions in deductive databases. These methods can be classified as either evaluation or optimization methods. Furthermore, the optimization methods can be classified into two categories: *query-dependent optimization* and *query-independent optimization.*

A *query-dependent* method rewrites a logic program into an equivalent but more efficiently evaluable one based on specific query forms. The magic rule rewriting technique is a typical example.

A *query-independent* method, on the other hand, compiles IDB predicates into a set of compiled forms independent of queries. When a query is submitted, an evaluation plan is generated based on the compiled form, current database statistics and the query information. The chain-based compilation and evaluation method is such a query-independent optimization method.

Although there have been some quite interesting studies on query-independent compilation of complex linear recursions ([HaZe92]) , these studies did not produce an efficient algorithm for automatic generation of compiled linear recursions. In this chapter, we discuss a simple variable connection graph matrix, *the V-matrix,* and develop a V-matrix expansion technique which discovers the minimal necessary expansions in the compilation of complex linear recursions[HaZe92]. Based on such V-matrix expansions, complex linear recursions can be normalized into highly regular *chain-forms* and *linear normal forms*. The compilation facilitates the development of powerful query analysis and evaluation techniques for complex linear recursions in deductive databases.

Definitions[Han89a] .

A **chain** of length k (k>1) is a sequence of k predicates with the following properties: (1) all k predicates have the same name, say p, and the l-th p of the chain is denoted as $p_{(l)}$, (2) there is at least one shared variable in every two consecutive predicates, and if the i-the variable in the first predicate is identical with the j-th variable in the second, the (i, j) is an invariant of the chain in the sense that the i-th variable of $p_{(l)}$ is identical with the j-th variable of $p_{(l+1)}$ for every l where $1 \leq l \leq k-1$. Each predicate of the chain is called a **chain predicate**. A chain predicate may consist of a sequence of connected non recursive predicates.

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 a **multi-chain recursion** when n >1. A recursion is **bounded** if it is equivalent to a set of non recursive rules.

A linear recursion is in **linear normal form** (LNF) if it consists of a set of exit rules and at most one *normalized recursive rule* in the form of:

$$r(X_1, X_2, ..., X_n) \text{ :- } c_1(X_1, Y_1), c_2(X_2, Y_2), ..., c_n(X_n, Y_n), r(Y_1, Y_2, ..., Y_n).$$

where X_i and Y_i ($1 \leq i \leq n$) are variable vectors. Each c_i ($1 \leq i \leq n$) is a **chain predicate**. Notice that a chain predicate ci for some i can be null in the sense that there is no ci predicate and $Y_i = X_i$. The **normalization** of a linear recursion is the process of transforming a linear recursion into its equivalent Linear Normal Form.

Normalization transforms linear recursions into highly regular compiled forms which not only facilitates systematic development of query analysis and evaluation techniques, but also helps binding propagation which may not be captured in other optimization methods.

Below is an example where the Magic Sets method encounters the problem of "lost bindings in propagation" while our normalization method can still obtain sufficient binding information [Han91b] .

Example 3-1. Suppose a query (q_1), " ?- r(X, Y, c)." is posed on the linear recursion :

$r(X, Y, Z)$:- $e_0(X, Y, Z)$. (r_0)

$r(X, Y, Z)$:- $a(X, Y), r(X_1, Z, Z_1), b(X_1, Z_1)$. ($r_1$)

Using the Magic Set rule rewriting technique, the goal node is adorned as r^{ffb}. The binding in the adorned goal node is propagated to the subgoal r in the body of the recursive rule, resulting in r^{fbf}, which is in turn propagated into the subgoal r in the body of the recursion at the next expansion, resulting in r^{fff}.

The resulting rule set is :

$r^{ffb}(X, Y, Z)$:- $r^{fbf}(X_1, Z, Z_1), a^{ff}(X, Y), b^{bb}(X_1, Z_1)$. ($r_{11}$)

$r^{fbf}(X_1, Z, Z_1)$:- $a^{fb}(X_1, Z), r^{fff}(X_2, Z_1, Z_2), b^{bb}(X_2, Z_2)$. ($r_{12}$)

Since the binding from the query could not be propagated further to the subgoal r^{fff} in r_{12}, the Magic Set involves the entire base relation a, thus the binding information cannot help reduce the set of data to be examined in Naive or Semi-Naive evaluation .

On the other hand, using the chain-based compilation method, the above recursion can be normalized into the following normalized form:

$r(X, Y, Z)$:- $e_0(X, Y, Z)$. (r_{21})

$r(X, Y, Z)$:- $a(X, Y), t(Z)$. (r_{22})

$t(Z)$:- $e_0(U, Z, V), b(U, V)$. (r_{23})

$t(Z)$:- $ab(Z, Z_1), t(Z_1)$. (r_{24})

where $ab(Z, Z_1)$:- $a(X_1, Z), b(X_1, Z_1)$.

After the compilation, the query binding "Z=c" can be propagated into every further expansion of the normalized recursion. Furthermore, the query(q_1) is essentially the verification of t(c) which can be evaluated efficiently by existence checking algorithm instead of evaluating the whole query closure. □

In this chapter, we discuss the concept and implementation of the compilation process, which automatically generates the chain-form and linear normal form of a function-free single linear recursion.

3.2 A Variable Connection Graph-Matrix: the V-matrix

The concept of chain-based compilation method was proposed in [Han89a]. Extensive study has been done on the compilation of complex linear recursions using a variable connection graph: the **V-graph**. The conclusion is that a single linear recursion is either a bounded recursion or can be compiled into highly regular chain forms.

Moreover, it is found that the expansion behaviour is closely related to its variable connection pattern. Based on further study of this discovery, the idea of **V-matrix** is introduced [HaZe92] and implemented [HaZeLu93] to automatically generate the compiled form of a linear recursion by simulating its expansion behaviour through V-matrix expansions.

To simplify our discussion, the following assumptions are made in this chapter:

1. The recursion is *single linear*, i.e. there is only one recursive rule in the cluster and it is a linearly recursive rule.
2. The rules are *function-free*, i.e. all the non-recursive predicates are EDB predicates.
3. There are no constant or duplicate variables appearing in the recursive predicate .
4. There is only one exit rule (non-recursive rule) in the recursive cluster, with the same head as the recursive rule, i.e. the exit rule (E_0) is of the form "$r(X_1, ..., X_n)$:- $e_0(X_1, ..., X_n)$".

Definition. [Han89a]

For a linear recursive rule with the head predicate $r(X_1, ..., X_n)$, the **0-th expansion** of r is defined as a tautological rule: "$r(X_1, ..., X_n)$:- $r(X_1, ..., X_n)$." the **first expansion** of r is the recursive rule of r. The **k-th expansion** of r ($k > 1$) is the unification of the recursive rule of r with the (k - 1)st expansion of r. The **k-th expanded exit rule** of r ($k \geq 0$), denoted as $e_k(X_1, ..., X_n)$ is the unification of the k-th expansion with the exit rule E_0.

The expansion behaviour of a recursion is closely related to the variable connections among its predicates.

Definition. [HanZe92]

Two predicates in the body of a rule are **connected** if they share a variable with each other or with a set of connected predicates. Two non-recursive predicates in the body of a rule are **U-connected** if they share a variable with each other or with a set of U-connected predicates. A set of variables are **U-connected** if they are in the same non-recursive predicate or in the same set of U-connected (non recursive) predicates.

The variables appearing in the recursive rule head are **distinguished variables**, while those appear only in the body of the rule are **non-distinguished variables**.

The variables of a recursive rule can be partitioned based on the relation of U-connections. In order to study the expansion behavior of a linear recursive rule, a variable connection graph-matrix, the V-matrix, is constructed as the following:

Definition. [HaZe92]

The variable connection graph-matrix, **V-matrix**, for a linear recursive rule of arity n consists of a sequence of rows. Each row consists of n columns with the i-th column corresponding to the i-th argument position of the recursive predicate. Moreover, there are possible U-connection edges between some columns in a row.

The contents of the initial V-matrix reflect the variable information in the original recursive rule, while the expansion of V-matrix simulates the expansion of the recursion rule.

The *initial V-matrix*, which consists of the first two rows (row[0] and row[1]) of the V-matrix, is constructed according to the following *V-matrix initialization rules*, while the expanded rows are constructed based on the V-matrix expansion rules to be discussed in next section.

V-matrix initialization rules [HaZe92] :

A V-matrix is initialized according to the following four steps:

1. Partition the variables in the rule according to the U-connections (each partition is called an U-connection set).
2. Copy the variables in the recursive predicate in the head and the body to the corresponding columns in row[0] and row[1] respectively.
3. Replace the variable at each column of row[1], say X, by the set of distinguished variables U-connected with X, if any.
4. Finally, set up a U-connection edge between each pair of columns in the corresponding row if the pair of columns are in row[0] and contain U-connected distinguished variables , or if they are in row[1] and contain U-connected nondistinguished variables. □

We will use the following set of rules throughout our discussion of this chapter[HaZe92]:

Example 3-2. The initial V-matrices of the recursive rules (A_1) to (G_1) are shown in Figure 3-1. (a) to (g).

$r(X)$:- $a(X, X_1), r(X_1)$. (A_1)

$r(X)$:- $a(X, X'), r(X_1)$. (B_1)

$r(X, Y)$:- $a(X, Y_1), r(X_1, Y_1), b(X_1, Y)$. ($C_1$)

$r(X, Y, Z)$:- $a(X, Y), r(X_1, Z, Z_1), b(X_1, Z_1)$. ($D_1$)

$r(X, Y)$:- $a(X, X_1, Y), b(Y, Y_1), r(X_1, Y_1)$. ($F_1$)

$r(X, Y, W, T, Z, U, V)$:- $r(Y, Y_1, T, T_1, Z_1, U_1, U_1)$,
$a(X, Y_1, T), b(W, T_1), c(Z), d(U, V, U_1)$. ($G_1$)

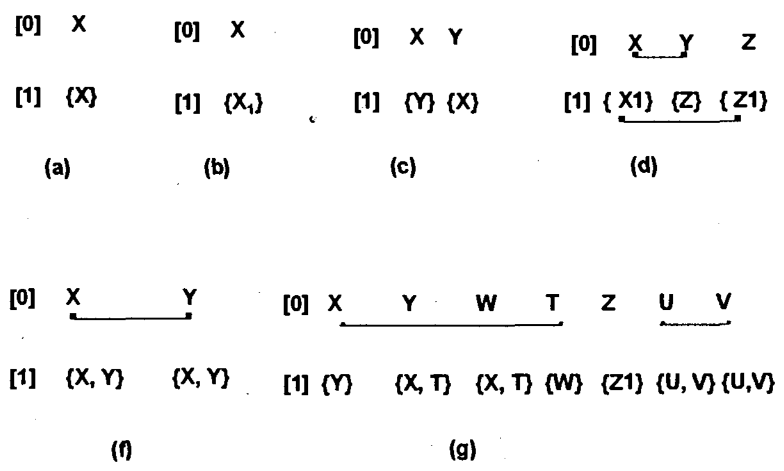


Figure3-1. The initial V-matrices of rules(A_1) to (G_1).

For rule(A_1), X_1 in row[1] is replaced by {X} because there is a U-connected set $\{X, X_1\}$. For rule(B_1), X_1 in row[1] retains X_1 because X_1 is not U-connected to any distinguished variable. For rule(C_1), X_1 is replaced by {Y} and Y_1 is replaced by {X} because there are two U-connected sets: $\{X, Y_1\}$ and $\{Y, X_1\}$. For rule(D_1), there are three U-connected sets: {X, Y}, {Z} and $\{X_1, Z_1\}$, but row[1] remains the same because none of the non-distinguished variables in row[1] is U-connected to any distinguished variables in row[0]. However, there are two U-connection edges, one between columns 1 and 2 in row[0] and the other between columns columns 1 and 3 in row[1]. For rule(F_1), there is one U-connection set : $\{X, Y, X_1, Y_1\}$ which contains all the variables, thus X_1 and Y_1 in row[1] are replaced by the set of distinguished variables {X, Y}. Moreover, there is one U-connection edge between columns 1 and 2 in row[0]. Finally, rule(G_1) contains six U-connection sets: $\{X, Y_1, T\}$, $\{W, T_1\}$, {Z}, $\{Z_1\}$, {Y} and $\{U, V, U_1\}$. Thus in row[1], Y_1 and T are replaced by {X, T}, T_1 by {T} and $\{U_1\}$ by {U, V}. Moreover, there are U-connection edges in row[0] : U-connection between columns (1, 4) and columns (6, 7). □

A V-matrix can be partitioned into one or more **unit V-matrices** based on the connections among V-matrix columns.

Definitions. [HaZe92]

Two columns of a V-matrix are **connected** if the two columns in the initial V-matrix share a variable, have a U-connection edge or are connected to a set of connected columns. A set of connected columns form a **unit V-matrix**. A linear recursive rule whose V-matrix consists of only one unit is a **single-unit rule**; otherwise it is a **multiple-unit rule**.

In example 3-2, the first five rules are single unit rules while the sixth rule (G_1) is a multiple-unit rule and its V-matrix consists of three units consisting of columns (1, 2, 3, 4), (5) and (6,7) respectively.

3.3 Derivation of Stable Rules by V-matrix Expansion

3.3.1 Expansion of Single-Unit Linear Recursive Rules

We first examine the expansion regularity of Single-Unit Linear Recursive Rules and the correspondence with its V-matrix expansion.

Example 3-3. The second expansion of rules (A_1) to (F_1) are (A_2) to (F_2) respectively [HaZe92]:

$r(X) \text{ :- } a(X, X_1), a(X_1, X_2), r(X_2).$ (A_2)

$r(X) \text{ :- } a(X, X'), a(X_1, X_1'), r(X_2).$ (B_2)

$r(X, Y) \text{ :- } a(X, Y_1), b(X_2, Y_1), r(X_2, Y_2), a(X_1, Y_2), b(X_1, Y).$ (C_2)

$r(X, Y, Z) \text{ :- } a(X, Y), a(X_1, Z), r(X_2, Z_1, Z_2), b(X_2, Z_2), b(X_1, Z_1).$ (D_2)

$r(X, Y) \text{ :- } a(X, X_1, Y), b(Y, Y_1), a(X_1, X_2, Y_1), b(Y_1, Y_2), r(X_2, Y_2).$ (F_2)

If each recursive rule generated at the second expansion is treated as an original recursive rule, row[2] of each V-matrix can be constructed as row[1] using the same V-matrix initialization rules.

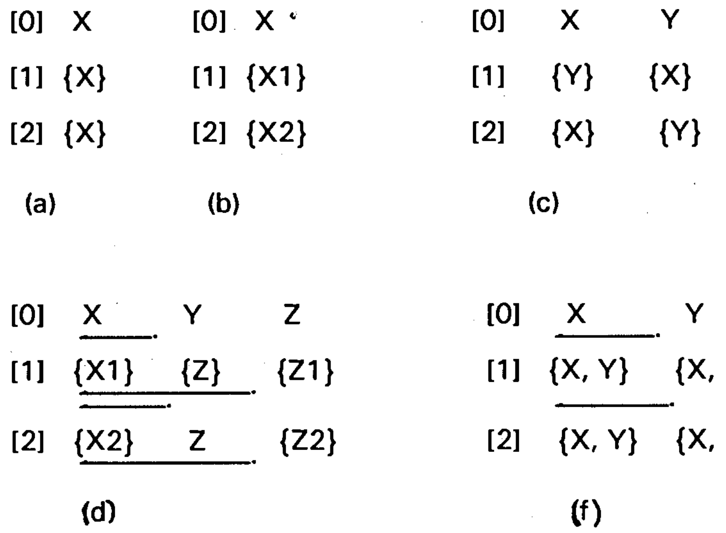


Figure 3-2. V-matrices of rules (A_1) to (F_1) at the second expansion.

Interestingly, row[2] of the V-matrices can also be derived from the initial V-matrix, i.e. from row[0] and row[1]. For example, Fig3-2(a) indicates that if a distinguished variable X at row[0] derives the same variable X at row[1], it will derive the same variable X at row[2]. Fig3-2(b) indicates that if a distinguished variable X at row[0] derives a non distinguished variable X_1 at row[1], it will derive a new non distinguished variable (in this case X_2) at row[2]. Fig3-2(c) indicates that if a distinguished variable X at row[0] derives another distinguished variable Y at row[1], it will derive the same distinguished variable Y from row[1] to row[2]. Fig3-2(d) is more complicated. According to the derivation rule observed from Fig3-2(b) and Fig3-2(c), row[2] should be [{X_2}, {Z_1}, {Z_2}]. However, by copying U-connection edges from row[1] to row[2] and then from row[0] to row[1], we can see that Z and Z_1 are now U-connected. Thus, Z_1 in row[2] is replaced by the distinguished variable Z and row[2] is now [{X_2}, {Z}, {Z_2}]. Similarly, row[2] of Fig3-2(f) is [{X, Y}, {X, Y}] because both X and Y derive the set {X, Y}.

□

This example shows that the expansion of V-matrix is closely related to the expansion of the recursive rule. In another words, the new rows of a V-matrix can be generated from its initial V-matrix by a set of V-matrix expansion rules, and the generated rows reflect the U-connectivity of the corresponding expanded recursive rules [HaZe92].

Definition.

A variable Y is a **derivative** of a distinguished variable X in a V-matrix if Y is derived by X, that is, Y and X are at the same column in the V-matrix, but Y's row number is X's row number +1.

V-matrix expansion rules [HaZe92] :

1. (Row generation) For each distinguished variable X in V-matrix [LastRow, i], add X's derivatives to V-matrix[NextRow, i].
2. (U-connection Propagation) The U-connection edges are copied from LastRow to NewRow and then from LastRow-1 to LastRow. If such copying makes a distinguished variable X U-connected to the set of variables in V-matrix[NewRow, i], X is added to the set of variables in V-matrix[NewRow, i].

Lemma 3-1.

Each row of the V-matrix generated by the above two V-matrix expansion rules correctly registers the set of distinguished variables U-connected to each column of the recursive predicate in the body at each expansion [HaZe92] .

In principle, a V-matrix can be expanded infinitely by following the V-matrix expansion rules. However, our goal is to find the regulation of its expansion. The theory of chain-based compilation [Han89a] says that "from a certain point, further expansions of a recursion rule will repeat the patterns of the existing expansion rules". In the representation of V-matrix, it means that from a certain row i, further expansions of a V-matrix will repeat the patterns of the existing rows in the V-matrix.

Definition. [HanZe92]

The **DV-set** (distinguished variable set) of a column is the set of all the distinguished variables U-connected to the variables in the column. Two rows, row[i] and row[j], in a V-matrix are **identical** (denoted as row[i] ≡ row[j]) if each pair of the corresponding columns have the same DV-set.

In Fig3-2, row[1] ≡ row[2] in (a) and (f). Similarly, row[2] ≡ row[1] in (b) and (d); row[2] ≡ row[0] in (c).

Obviously, if row[j] ≡ row[i] (j > i), all the expansions starting at row[j] repeats the expansions starting at row[i]. Thus we have :

Lemma 3-2. [HaZe92]

In a single-unit V-matrix, if row[i] ≡ row[j], and i < j, then row[j+k] ≡ row[i+k] for any k ≥ 0.

Based on such regularity of V-matrix expansions, we define the *stable level* and the *period* of a V-matrix as:

Definition.

If starting at row S, there exists a T such that the row of a single-unit V-matrix repeats at every T more expansions, that is, row[S+k*T] ≡ row[S] for all k > 0, then S is the **stable level** and the smallest T the **period** of the V-matrix. If row[S] contains no distinguished variables, T is defined as 0.

Algorithm 3-1. The expansion of a single-unit V-matrix and the derivation of its stable level S and period T [HaZe92] .

Input: An initial single-unit V-matrix.

Output: An expanded V-matrix, the stable level S and the period T.

Method:

```
begin
 LastRow := 0; CurrentRow := 1;
 while not RowRepeating (CurrentRow, ExistingRow)
   do
     begin
           LastRow := CurrentRow;
           CurrentRow := CurrentRow + 1;
           /* Generate the contents of the CurrentRow. */
           for each column  i  do
                  /* Every column in CurrentRow is initially empty. */
                  for each  distinguished variable  X in V-matrix[LastRow, i]  do
                     Add x's derivatives to V-matrix[CurrentRow, i];
                     /* U-connection Propagation. */
                     Copy the U-connections from LastRow to CurrentRow;
                     Copy the U-connections from LastRow - 1 to LastRow;
                       for each column i do
                         for each X in V-matrix[CurrentRow, i]  do
                           if X is U-connected to a distinguished variable Y which
                           is not already in V-matrix[CurrentRow, i]
                              then Add Y to V-matrix[CurrentRow, i]
                                  and remove, if any, non distinguished variables there;
     end;
   S := ExistingRow;
   if there is no distinguished variable in CurrentRow
   then  T := 0
   else  T := CurrentRow - ExistingRow
 end. □
```

Notice that *RowRepeating* is a Boolean function which returns true if there is an ExistingRow, where $0 \leq$ ExistingRow < CurrentRow, such that row[ExistingRow] $\equiv$ row[CurrentRow]. That is,

```
function RowRepeating (CurrentRow,  var ExistingRow): Boolean;
  begin
     ExistingRow := CurrentRow - 1;
     repeat
          If row[CurrentRow] ≡ row[ExistingRow]
            then  return(true)
            else ExistingRow := ExistingRow - 1
     until ExistingRow < 0;
     return(false)
  end. □
```

For Example 3-2, the stable levels and periods of the recursions (A_1) to (F_1) are shown in Table3-1.

Rule	A1	B1	C1	D1	F1
S	0	1	0	1	0
T	1	0	2	1	1

Table3-1. Stable levels and periods of the recursions A_1 to F_1.

Theorem 3-1. In a single-unit recursive rule of arity n, the expansion of its V-matrix terminates at or before the n-th iteration. That is, $S+T \leq n$ [HaZe92].

3.3.2 Expansion of Multiple Unit Linear Recursive Rules

Although the V-matrix expansion rules can be applied directly to multiple-unit linear recursions, it may lead to relatively large number of expansions. Suppose there are k units in the V-matrix, the minimum number of expansions to reach the row repeating stage should be the least common multiplier of the arity of each unit, that is, $\mathrm{lcm}(n_1, n_2, \ldots, n_k)$, where n_j $(1 \leq j \leq k)$ is the arity of the j-th unit.

For example, if a V-matrix consists of three unit matrices V_1, V_2 and V_3, with $S_1=S_2=S_3=0$, $T_1=5$, $T_2=6$, and $T_3=7$, it will take $T = \mathrm{lcm}(5,6,7) = 210$ expansions to find a repeating row in the combined V-matrix.

However, since each V-matrix unit reaches its own stable stage independent of other units, the stability of a recursion can be determined by the examination of each unit independent of others. That is, for the above example, only 7 expansions are necessary to detect the regularity of the expansion patterns.

Definition. [HaZe92]

A linear recursion whose recursive rule corresponds to a multiple unit V-matrix reaches a **stable stage** at the S-th expansion if for every unit V-matrix V_i, either no new results can be generated from this unit on any EDBs (i.e. bounded for this unit), or each further T_i expansion adds to the body of the rule the same set of EDB predicates connected to the same set of variables in the unit.

For the recursive rule (G_1) in Example 3-2, we have three unit V-matrices. According to Algorithm 1, we have the S and T for each of them as follows: $S_1=1$, $T_1=2$; $S_2=1$, $T_2=0$; $S_3=0$, $T_3=1$. Therefore, their combined stable level is S = maximum(S_1, S_2, S_3)=1, with $T_1=2$, $T_2=0$ and $T_3=1$.

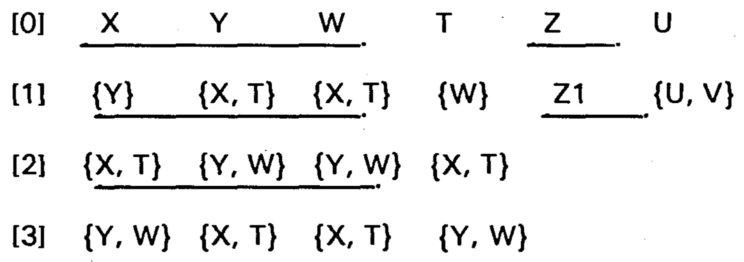

[0] X Y W T Z U
[1] {Y} {X, T} {X, T} {W} Z1 {U, V}
[2] {X, T} {Y, W} {Y, W} {X, T}
[3] {Y, W} {X, T} {X, T} {Y, W}

Figure 3-3. The expansion of the V-matrix of rule (G_1)

3.4. Compilation of Linear Recursions

The goal of the compilation phase is to generate *chain forms* and *linear normal forms* (LNF) of a linear recursion based on the expansion of its V-matrix. For simplicity, we discuss only the case of *single-unit recursions*, the chain form and linear normal form of a multiple-unit recursion can be derived by merging the compiled forms of all its units. Detailed discussion on the compilation of multiple-unit recursions can be found in [HaZe92].

3.4.1 Automatic Generation of Chain Forms for Single-Unit Linear Recursions

Since the variable connectivity of an expanded V-matrix corresponds to the variable connectivity of the expanded recursive rules, the regularity of the expansions of a V-matrix corresponds to the regularity of the expansion of its corresponding recursion. Thus, "when a single-unit V-matrix reaches its stable level, the corresponding linear recursion reaches its stable stage" [HaZe92].

Definition. [HaZe92]

A linear recursion is **stable** at the S-th expansion if (1) it generates no new results for any EDBs by further expansions (*bounded recursion*), or (2) it adds to the body of an expanded rule the same set of EDB predicates U-connected to the same set of distinguished variables at every further T expansions (*periodicity*).

Lemma 3-3. A single-unit linear recursion is bounded if its period T = 0. [HaZe92]

In Example 3-2, the recursion (B_1) is bounded.

Lemma 3-4. The number of potential chains of a single-unit linear recursion is the number of distinct DV-sets in the row [S+T] of its V-matrix, where S is the stable level and T is the period of the V-matrix [HaZe92] .

In Example 3-2, the [S+T] row of the V-matrices indicates that (A_1) has one potential chain, (C_1) has two potential chains , (D_1) has one potential chain and (F_1) has one potential chain.

Each **chain** is a set of U-connected predicates that (1) repeats at every T expansions starting at the S-th expansion, and (2) connects the columns corresponding to the DV-set in row[S] to row[S+k*T].

Basically a chain acts like a bridge that connects the corresponding *chain variable positions* between expansions S and S+K*T. The predicates in a chain must satisfy two **connectivity conditions**: (1) The U-connection set of all the predicates in a chain must be connected to the chain position variables on the Sth and S+Tth expansions. (2) The predicates in a chain must be U-connected to each other.

The Chain Generating Rules:

1. **Choose the set of chain predicates for each chain.**

   The set of non-recursive predicates generated from the (S+1)-th expansion to the (S+T)-th expansion forms the "**candidate set**" of the chain predicates. This is because the same set of predicates repeats at each T further expansion. However, if the candidate set of a chain does not satisfy either of the two connectivity conditions, some of the predicates in the candidate set should be replaced by their corresponding predicates in earlier expansions. The result is called the "**primitive set**", which is the actual set of chain predicates.

2. **Properly index the chain variables.**

   The chain variables need to be indexed properly to reflect the *expansion regularity*. There are two general rules to form the head of a chain predicate:

   - The variables not shared with any predicates outside of the chain should not appear in the chain predicate. The reason is that these variables are only "internal" to the chain, they do not participate in binding information exchange between chains during iterative processing.
   - The remaining variables should be indexed properly to reflect the information passing between iterations. Considering the chain position variables in the recursive predicate of the S-th and (S+T)-th expansions, and name them the **S-set** and **ST-set** respectively. For each variable in a chain predicate, the variable in the ST-set should have the same variable name as the corresponding variable in the S-set but with the index increased by one. Some variables may be in both S-set and ST-set, in such cases we extract a "**thread**" of variables and name them according to the position in the thread. For example, if the S-set is [A, B, C] and the ST-set is [B, C, D], we find a thread "$A \rightarrow B \rightarrow C \rightarrow D$", thus the variables A, B, C, D should be renamed as X_i, X_{i+1}, X_{i+2}, X_{i+3} respectively.

Definition. [HaZe92]

If the set of the variables at the (S+T)-th expansion is the same as the set of distinguished variables of the head predicate, the corresponding potential chain is trivially true and the chain is a "**null chain**".

Lemma 3-5. Following the chain construction rules, the chain predicates and their associated variable names and indices are generated correctly [HaZe92] .

Algorithm 3-2. Generation of the compiled chain form for a single-unit linear recursion.

Input: A linear recursion r, its expanded V-matrix and its stable level S and period T.

Output: The compiled chain form of recursion R.

Method:

Case 1.T = 0.

The recursion r is bounded and the compiled form is the union of the expanded exit rules from 0-th to S-th expansions. That is:

$$\begin{aligned} r(U_1, V_1, ..., X_1) &= E_0(U_1, V_1, ..., X_1) \cup E_1(U_1, V_1, ..., X_1) \cup ... \\ &\cup E_S(U_1, V_1, ..., X_1). \end{aligned}$$

Case 2. T≠0, but the recursion contains only null chains.

The recursion r is bounded and its compiled form is the union of the k-th expanded exit rules for $0 \leq k \leq S+T-1$. That is:

$$\begin{aligned} r(U_1, V_1, ..., X_1) &= e_0(U_1, V_1, ..., X_1) \cup e_1(U_1, V_1, ..., X_1) \cup ... \\ &\cup e_{S+T-1}(U_1, V_1, ..., X_1). \end{aligned}$$

Case 3. $T \neq 0$ and the recursion contains some non-null chains.

The recursion is a single-chain or multiple-chain recursion with the following compiled form: $R = SS \cup (\cup_{i=0 \text{ to } \infty} (MM, CC^i, TT))$,

which consists of four portions: (i) pre-stable exit rule portion (SS), (ii) stable exit rule portion (TT), (iii) chain-portion (CC) and (iv) miscellaneous portion (MM).

The **SS-portion** represents the pre-stable expansions, which is the union of E_k for k from 0 to S-1 if $S > 0$ or empty otherwise. That is,

$$SS = \cup_{i=0 \text{ to } S-1} e_i (U_1, V_1, \ldots, X_1).$$

The **TT-portion** consists of the bodies of the exit rules contributing to the period of the recursion, which is the union from e_0 to e_{T-1}. That is:

$$TT = \cup_{j=0 \text{ to } T-1} e_j(U_i, V_i, \ldots, X_i).$$

The **CC-portion** consists of a set of non-null chains, represented by its chain form in the i-th iteration. Each non-null chain predicate is generated following the chain generating rules.

Finally the **MM-portion** is composed by the non-recursive predicates in the body of the (S+T)-th expansion which are not chain predicates. Notice that the variables in these predicates need to be adjusted if they are chain predicates, because such variables could be involved in the chain in the next expansion. □

Example 3-4. Generating compiled chain forms of (A_1) to (F_1).

For (A_1), we have S=0 and T=1. The S-th expansion and (S+T)-th expansions are :

S-rule: r(X) :- r(X).

ST-rule: r(X) :- a(X, X_1), r(X_1).

Obviously, there is one chain and the chain predicate set is simply { a(X, X_1) } which connects r(X) and r(X_1). Thus, the CC portion is a(X_{i-1}, X_i), the TT portion is $e_0(X_i)$ while SS portion and MM portion are ∅. Therefore , the compiled chain form for A_1 is: $r(X_0) = \cup_{i=0 \text{ to } \infty}(a^i(X_{i-1}, X_i), e_0(X_i))$.

For (B_1), we have S=1 and T=0. The recursion is bounded, and the complied form is:

$r(X) = e_0(X) \cup e_1(X) = e_0(X) \cup (a(X, X'), e_0(X_1))$.

For (C_1), we have S=0 and T=2. The S-th and (S+T)-th expansions are:

S-rule: r(X, Y) :- r(X, Y).

ST-rule: r(X, Y) :- a(X, Y_1), a(X_1, Y_2), r(X_2, Y_2), b(X_1, Y), b(X_2, Y_1).

The expanded V-matrix [Fig3-2(c)] shows that there two DV-sets in row[S+T]: {X} and {Y}, thus there are potentially two chains.

Chain1 connects variables on column[0] of the recursive predicates in the S-th and (S+T)-th expansion, i.e. X and X_2, thus we have chain1: "ab(X, X_2) :- a(X, Y_1), b(X_2, Y_1)." Chain2 connects variables on column[1] of the recursive predicates in the S-th and (S+T)-th expansion, i.e. Y and Y_2, thus we have chain2: "ba(Y, Y_2) :- b(X_1, Y), a(X_1, Y_2)."

The SS-portion and MM-portion are $\varnothing$, and TT-portion is $e_0(X_i, Y_i) \cup e_1(X_i, Y_i)$ based on variable renaming.

The compiled chain form for C_1 is:

$r(X_0, Y_0) = \cup_{i=0 \text{ to } \infty} (ab^i(X_{i-1}, X_i), ba^i(Y_{i-1}, Y_i), (e_0(X_i, Y_i) \cup e_1(X_i, Y_i)))$.

For (D_1), we have S=1, T=1. Its S-th and (S+T)-th expansions are:

S-rule:	$r(X, Y, Z)$:- $a(X, Y), r(X_1, Z, Z_1), b(X_1, Z_1)$.
ST-rule:	$r(X, Y, Z)$:- $a(X, Y), a(X_1, Z), r(X_2, Z_1, Z_2), b(X_2, Z_2), b(X_1, Z_1)$.

There is only one DV-set in row[S+T] of the matrix, thus there is one potential chain. The chain should connect the variables on column[1] of the recursive predicate in the body of S-rule and ST-rule, i.e. Z and Z_1.

The *candidate set* of the chain is $\{a(X_1, Z), b(X_2, Z_2)\}$ because they are newly generated in the 2nd expansion. However, $b(X_2, Z_2)$ is not U-connected to $a(X_1, Z)$, neither is it connected to the chain variables Z or Z_1, thus it needs to be "swapped" with its corresponding predicate in the previous expansions : $b(X_1, Z_1)$. The new set of chain predicates $\{a(X_1, Z), b(X_1, Z_1)\}$ satisfies both connectivity conditions and is thus the *primitive set* of the chain. The chain is : "$ab(Z, Z_1)$:- $a(X_1, Z), b(X_1, Z_1)$."

The SS-portion is $e_0(X, Y, Z_0)$, the CC-portion is $ab(Z_i, Z_{i+1})$, the TT-portion is $e_0(U, Z_i, V)$ and the MM-portion is $a(X, Y), b(U, V)$.

The compiled chain form of D_1 is:

$$r(X, Y, Z_0) = e_0(X, Y, Z_0) \cup \cup_{i=0 \text{ to } \infty} (a(X, Y), b(U, V), ab^i(Z_{i-1}, Z_i), e_0(U, Z_i, V)).$$

For (F_1), we have S=0 and T=1. The S-rule and ST-rule are:

S-rule: $r(X, Y)$:- $r(X, Y)$.

ST-rule: $r(X, Y)$:- $a(X, X_1, Y), b(Y, Y_1), r(X_1, Y_1)$.

There is one DV-set {X, Y} in row[S+T], thus it has one potential chain. The chain is obviously "$ab(X, Y, X_1, Y_1)$:- $a(X, X_1, Y), b(Y, Y_1)$."

Thus, the chain form of (F_1) is:

$$r(X_0, Y_0) = \cup_{i=0 \text{ to } \infty} (ab^i(X_i, Y_i, X_{i+1}, Y_{i+1}), e_0(X_i, Y_i)). \square$$

3.4.2. Normalization of Single-Unit Linear Recursions

Although the internal processing of a linear recursion is largely based on its chain form, it is not very obvious when comparing to other rule rewriting methods, such as the Magic Sets method. However, the compilation method based on V-matrix expansions can be viewed alternatively as a query independent rule rewriting process, which transforms a complex linear recursion into a normalized linear recursion. Furthermore, the generation of the linear normal form from its corresponding chain form is quite straightforward.

The compiled chain form of a linear recursion is either a bounded recursion or an n-chain recursion. A bounded recursion is already in linear normal form since it consists of only non-recursive rules. A complied n-chain recursion can be transformed into linear normal form composed of the following set of rules [HaZeLu93] :

1. A set of *exit rules* in the form of "r :- S.", where s is a disjunct in the SS-portion of the compiled chain form;

2. One *auxiliary rule* in the form of "r :- M, P.", where m is the set of predicates in the MM-portion which do not share variables with the TT-portion at the i-th expansion, and p is an auxiliary predicate;
3. One *normalized linear recursive rule* in the form of "$p(X_1, X_2, ..., X_n)$:- $c_1(X_1, Y_1)$, $c_2(X_2, Y_2)$, ..., $c_n(X_n, Y_n)$, $p(Y_1, Y_2, ..., Y_n)$. " , where c_1, ..., c_n is a set of chain predicates which form the CC-portion of the compiled form; and
4. A set of *stable exit rules* in the form of "p :- M', T.", where T is a disjunct in the TT-portion of the compiled chain form, and M' is the set of remaining predicates in the MM-portion. □

Example 3-5. Transforming recursions (A_1) to (F_1) into linear normal form :

For (A_1), the recursive rule is already in normalized form. Thus the linear normal form of the recursion is:

$r(X)$:- $e_0(X)$.

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

For (B_1), it is a bounded recursion, thus its LNF is :

$r(X)$:- $e_0(X)$.

$r(X)$:- $a(X, X')$, $e_0(X_1)$.

For (C_1), the recursion consists of two chains, and its SS-portion and MM-portion are ∅, thus the linear normal form is:

r(X, Y) :- e_0(X, Y).

r(X, Y) :- e_1(X, Y).

r(X, Y) :- ab(X, X_1), ba(Y, Y_1), r(X_1, Y_1).

where e_1(X,Y) is the first expanded exit rule "e_1(X, Y) :- a(X, Y_1), r(X_1, Y_1), b(X_1, Y)."; ab(X, X_1) and ba(Y, Y_1) are two chain rules "ab(X, X_1) :- a(X, T), b(X_1, T)." and "ba(Y, Y_1) :- a(T, Y_1), b(T, Y). "

For (D_1'), it is a single recursion, and its MM-portion consists of two parts: MM_1 = {a(X, Y)} which does not share variables with the TT-portion and MM_2 = {b(U, V)} which shares variables with the TT-portion. Thus, its linear normal form is:

r(X, Y, Z) :- e_0(X, Y, Z).

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

p(Z) :- e_0(U, Z, V), b(U, V).

p(Z) :- ab(Z, T), p(T).

Note: (D_1) is the same recursion in Example3-1, we already see that for some query forms, the Magic Sets method may lose the binding information in its propagation, but if the rules are rewritten into linear normal form, it can capture the binding information properly and pass it onto further iterations.

For (F_1), the recursion is a single chain recursion and its linear normal form is:

r(X, Y) :- e_0(X, Y).

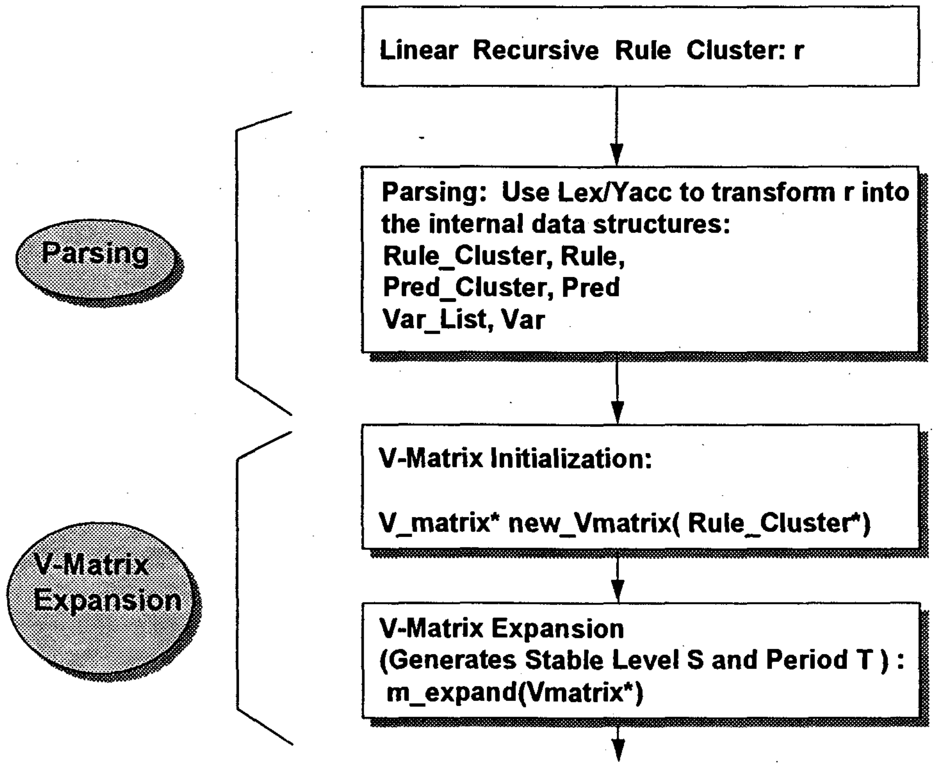
r(X, Y) :- ab(X, Y, X_1, Y_1), r(X_1, Y_1).

where "ab(X, Y, X_1, Y_1) :- a(X, X_1, Y), b(Y, Y_1)." and the variables < X, Y > and < X_1, Y_1 > form two variable vectors. ◫

3.5 Implementation of the Chain-Based Compilation Method

So far, we have discussed the concept and algorithms of the chain-based compilation method. One of the major contributions of this thesis is the implementation of the compilation process discussed above. The compilation subsystem parses and compiles a complicated single linear recursive rule cluster into a highly regular chain form, which provides the basis for further analysis and evaluation [Chapter 4].

This section summarizes the implementation and discusses a compilcated example to illustrate the process. Below is the flowchart of the overall design:


Linear Recursive Rule Cluster: r
Parsing
Parsing: Use Lex/Yacc to transform r into the internal data structures: Rule_Cluster, Rule, Pred_Cluster, Pred Var_List, Var
V-Matrix Expansion
V-Matrix Initialization: V_matrix* new_Vmatrix(Rule_Cluster*)
V-Matrix Expansion (Generates Stable Level S and Period T) : m_expand(Vmatrix*)

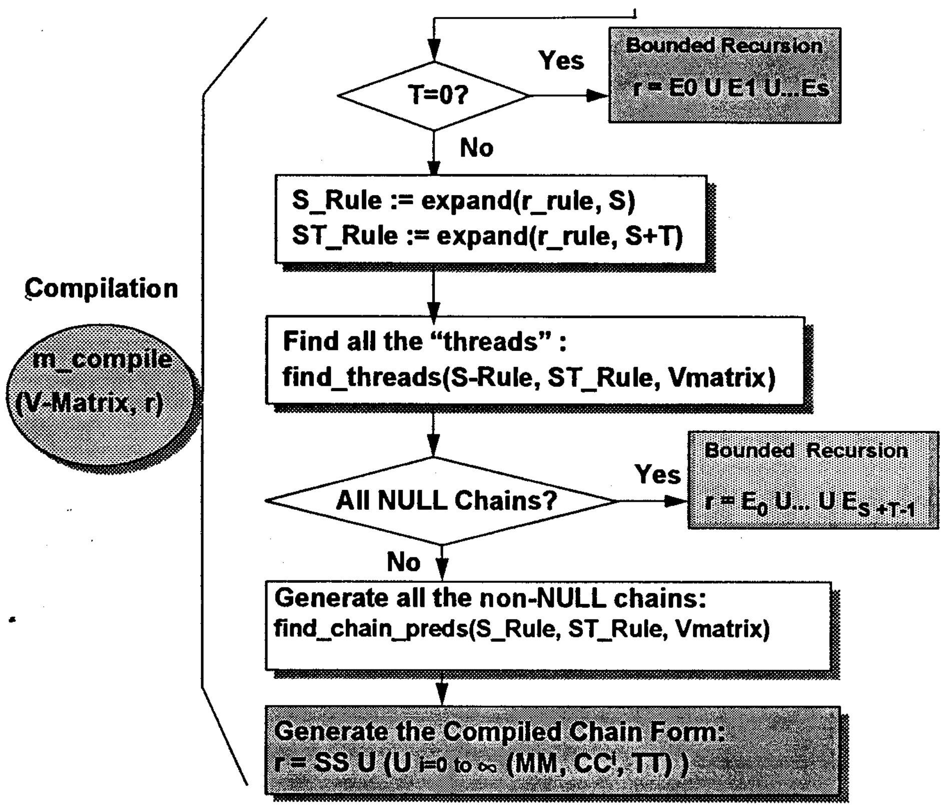

Compilation
m_compile (V-Matrix, r)
T=0?
Yes
Bounded Recursion r = E0 U E1 U...Es
No
S_Rule := expand(r_rule, S)
ST_Rule := expand(r_rule, S+T)
Find all the "threads" : find_threads(S-Rule, ST_Rule, Vmatrix)
All NULL Chains?
Yes
Bounded Recursion r = E0 U... U ES+T-1
No
Generate all the non-NULL chains: find_chain_preds(S_Rule, ST_Rule, Vmatrix)
Generate the Compiled Chain Form: r = SS U (U i=0 to ∞ (MM, CCi, TT))

Figure 3-4. The compilation flow chart

The implementation consists of three major phases:

1. The *parsing phase* uses Lex/Yacc to parse the rule cluster and transforms it to internal data structures for further processing.
2. The *V-matrix expansion phase* implements the V-matrix initialization rules, V-matrix expansion rules and Algorithm3-1, which initializes and expands the V-matrix of the recursive rule , and generates the *stable level S* and *period T*.
3. The *compilation phase* is based on Algorithm3-2, which generates the compiled chain forms of the recursion. The main procedure of this part, find_chain_preds, is further broken down into three parts:

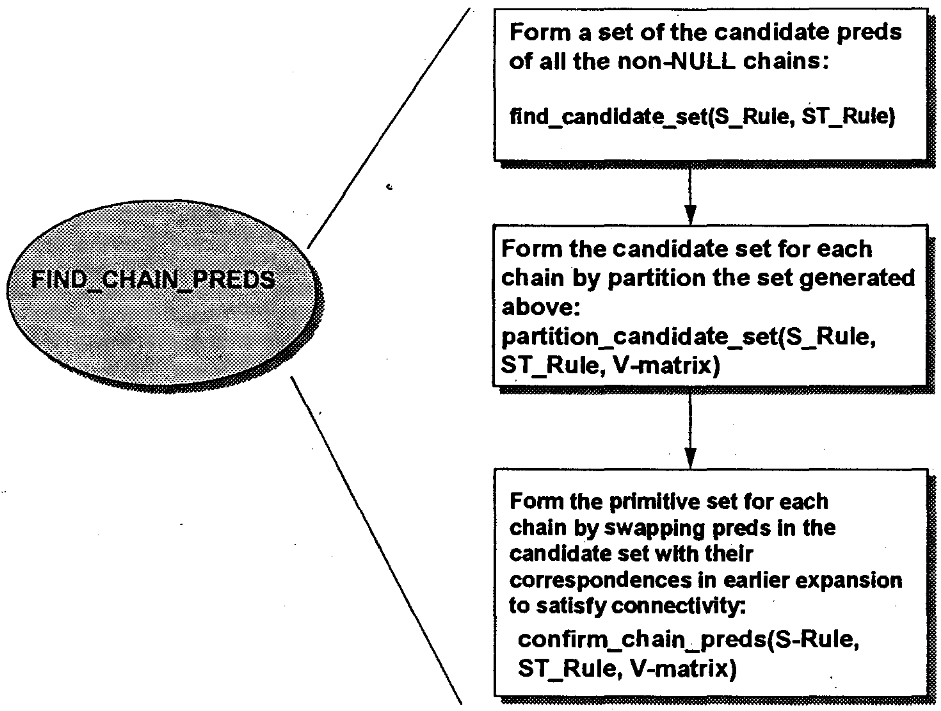

FIND_CHAIN_PREDS
Form a set of the candidate preds of all the non-NULL chains:
find_candidate_set(S_Rule, ST_Rule)
Form the candidate set for each chain by partition the set generated above:
partition_candidate_set(S_Rule, ST_Rule, V-matrix)
Form the primitive set for each chain by swapping preds in the candidate set with their correspondences in earlier expansion to satisfy connectivity:
confirm_chain_preds(S-Rule, ST_Rule, V-matrix)

Figure 3-5. The chain generating process.

The data structure "**chain_form**" for a k-chain recursion is defined as follows:

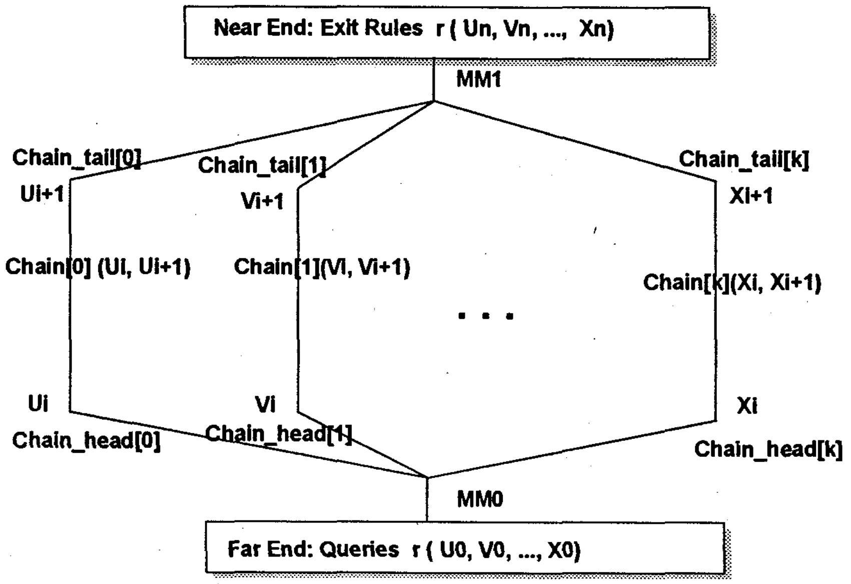


Figure 3-6. The internal data structure: Chain_Form

The compilation process is isllustrated by analyzing the following example:

Example 3-6.

r(A, B, C, D, E) :- a(A, B), b(C, E), c(F, D), d(G, H), r(C, F, D, G, H). (r_1)

r(A, B, C, D, E) :- e(A, B, C, D, E). (r_2)

Phase 1. Parsing.

The parsing phase reads in the two rules and forms a **rule set**. Notice that a set is different from a list in that elements in a set are unordered. Thus, the order in which these two rules are specified is not relevant to the processing part. Similarly, the body of each rule is defined internally as a *set* of predicates, while the variables of each predicate form a *list.*

Phase 2. V-matrix Expansion.

Using the V-matrix initialization and expansion rules [3.2, 3.3], we have the following expanded V-matrix for r_1:

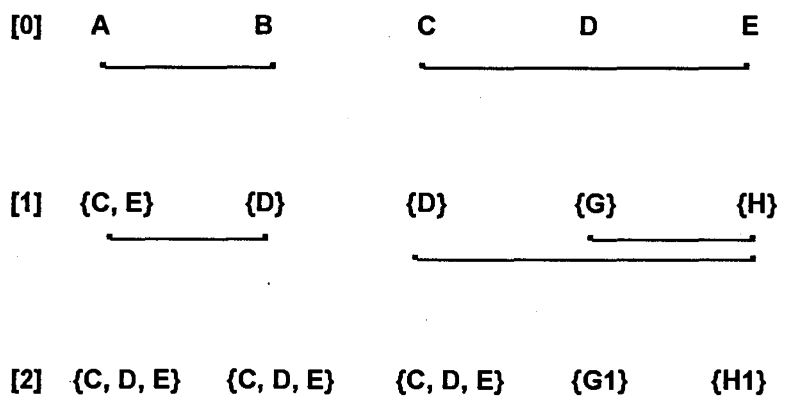

[0] A B C D E
[1] {C, E} {D} {D} {G} {H}
[2] {C, D, E} {C, D, E} {C, D, E} {G1} {H1}

Figure 3-7. The expanded V-matrix of Example 3-6.

Considering the U-connection edges, row[1] ≡ row[2]. Thus, we get the stable level S = 1 and the period T = 1.

Phase 3. Generate the compiled chain form.

Since $T \neq 0$, we generate the S-th and (S+T)-th expansion of the recursive rule:

S_Rule:	r(A, B, C, D, E)	:-	a(A, B), b(C, E), c(F, D), d(G, H),
			r(C, F, D, G, H).
ST_Rule:	r(A, B, C, D, E)	:-	a(A, B), b(C, E), c(F, D), d(G, H),
			a(C, F), b(D, H), c(F_1, G), D(G_1, H_1),
			r(D, F_1, G, G_1, H_1).

By checking the row[S+T] of the expanded V-matrix, we find that there is only one set of *distinguished variables*, i.e. {C, D, E}, which means that there is only one potential chain for this recursion [Lemma 3-4]. The columns that this variable set resides indicate the chain variable positions, which is columns 1, 2 and 3 in this case.

Next, we extract the "thread" of the chain, i.e. the variable connection pattern from the recursive predicate in the body of the S_Rule to that of the ST_Rule. In this case, the corresponding variable sets are [C, F, D] and [D, F_1, G]. Thus, we have two "threads", i.e. "$C \rightarrow D \rightarrow G$" and "$F \rightarrow F_1$". Since the threads are not cyclic, this is a non-NULL chain. Thus, we will generate the chain form according to Algorithm 3-2.

The *candidate set* of the chain predicates consists of the predicates generated from the S-th to (S+T)-th expansion, i.e. "a(C, F), b(D, H), c(F_1, G), d(G_1, H_1)". However, this set does not satisfy the second connectivity condition, because none of the predicates in this set is U-connected to the others. Thus, some or all of the predicates in the candidate set must be exchanged (or swapped) with their corresponding predicates generated in earlier expansions.

We swap $c(F_1, G)$ with $c(F, D)$, and $d(G_1, H_1)$ with $d(G, H)$. The new set of potential chain predicates is now "a(C, F), b(D, H), c(F, D), d(G, H)", which is (1) connected to the chain variable sets {C, F, D} and {D, F_1, G}, and (2) U-connected among its elements, because there exists a U-connection chain , "a(C, F) $\leftrightarrow$ c(F, D) $\leftrightarrow$ b(D, H) $\leftrightarrow$ d(G, H)", which connects all the member predicates. Thus, this is the *primitive set* of the chain predicates.

After examing which variables should appear in the chain definition, we have the chain defined as :

abcd(C, D, G) :- a(C, F), b(D, H), c(F, D), d(G, H).

Notice that, only the chain predicates {C, D, G} appears outside of the chain in the (S+T)-th expansion, all other variables are internal to the chain.

After variable renaming and indexing, we have the final chain form defined as :

$abcd(X_{i-1}, X_i, X_{i+1})$:- $a(X_{i-1}, F)$, $b(X_i, H)$, $c(F, X_i)$, $d(X_{i+1}, H)$.

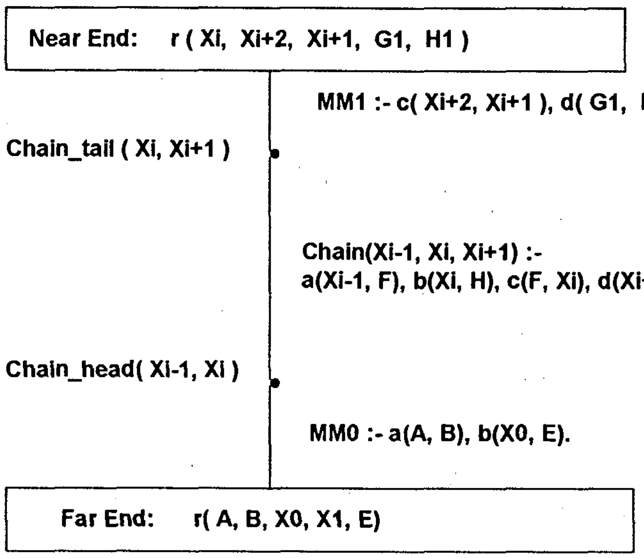
Thus, the compiled form is [Algorithm 3-2]:

$r(A, B, X_0, X_1, E) = e(A, B, X_0, X_1, E) \cup (\bigcup_{i=0 \text{ to } \infty} (a(A, B), b(X_0, E), abcd^i(X_{i-1}, X_i, X_{i+1}), c(X_{i+2}, X_{i+1}), d(G_1, H_1), e(X_i, X_{i+2}, X_{i+1}, G_1, H_1))$.

The correctness of the compilation can be verified by comparing the expanded exit rules with the corresponding rules generated by the chain form.

For example, when i=0, the rule generated by the chain form is "r(A, B, X_0, X_1, E) :- a(A, B), b(X_0, E), c(X_2, X_1), d(G_1, H_1), e(X_0, X_2, X_1, G_1, H_1)". It is the same as the first expanded exit rule. When (i=1), the rule generated by the chain form is "r(A, B, X_0, X_1, E) :- a(A, B), b(X_0, E), abcd(X_0, X_1, X_2), c(X_3, X_2), d(G_1, H_1), e(X_1, X_3, X_2, G_1, H_1)", which is the second expended exit rule. When (i=2), the rule generated by the chain form is "r(A, B, X_0, X_1, E) :- a(A, B), b(X_0, E), abcd(X_0, X_1, X_2), abcd(X_1, X_2, X_3), c(X_4, X_3), d(G_1, H_1), e(X_2, X_4, X_3, G_1, H_1)", which is the third expanded exit rule, etc.

To summarize, the rule generated at the i-th iteration of the chain form is the (i+S)-th expanded exit rule, and the exit rules before the stable stage are covered by the SS portion of the compiled form. Thus, the compilation is complete. Moreover, by compiling the recursion into chain forms, the information passing between iterations is made clear, which greatly facilitates systematic analysis [Chapter 4]. Below is the compiled chain form represented by the internal data structure "Chain_Form" :


Near End: r (Xi, Xi+2, Xi+1, G1, H1)
MM1 :- c(Xi+2, Xi+1), d(G1, H1).
Chain_tail (Xi, Xi+1)
Chain(Xi-1, Xi, Xi+1) :-
a(Xi-1, F), b(Xi, H), c(F, Xi), d(Xi+1, H).
Chain_head(Xi-1, Xi)
MM0 :- a(A, B), b(X0, E).
Far End: r(A, B, X0, X1, E)

□

Chapter 4
Evaluation of Functional Nested Linear Recursions

In Chapter 3, we discussed how to compile a function-free single linear recursion into chain forms and linear normal forms. In this chapter, we will discuss the query evaluation process based on the compiled forms and the extension of the chain-based compilation and evaluation method to the domain of functional nested linear recursions.

The major contribution of this part is the extension of the compilation, analysis and evaluation algorithms to the domain of *functional nested linear recursions*, which will be illustrated by a detailed discussion of the n-queens problem[HaLu92]. However, for the purpose of completeness and ease of understanding, the algorithms on the analysis and evaluation of functional recursions are also included [4.1,4.3,4.4]. Detailed studies in these areas are provided in [Han92, Han91a and HaWa91], the discussion in this Chapter focuses on implementation aspects.

4.1. From Function Free to Functional Recursions

In the discussion of Chapter 3, the assumption was made that all base relations are finite EDB relations. However, in many real life deductive database applications, functions (especially relational, arithmetic and list functions) are encountered frequently. The airline information system (Example 2-6) is one such application. Because of this, we need to generalize the applicable domain of the chain-based compilation and evaluation method to include **functions** in the rule definition and query processing.

Example 4-1. A typical linear recursion with list functions, append(L_1, L_2, L_3) is defined as:

append([], L, L). (r_1)

append([X | L_1], L_2, [X | L_3]) :- append(L_1, L_2, L_3). (r_2)

We will use this example extensively to illustrate the analysis and evaluation of functional recursions.

To compile such functional recursions, we need two preprocessing steps to rewrite the rules:

(1) **function-predicate transformation**: This maps a function together with its functional variable to a predicate (called *functional predicate*), where the functional variable is the variable which unifies the returned value(s) of the function.

That is, each function of arity n is transformed to a predicate of arity n+1, with the last argument representing the unifying variable.

For Example 4-1, the rules (r_1) and (r_2) are rewritten as (r_1') and (r_2') after the function-predicate transformation:

append(Y, L, L) :- Y = []. (r_1')

append(U, L_2, W) :- append(L_1, L_2, L_3), cons(X, L_1, U), cons(X, L_3, W). (r_2')

(2) **rectification**: Logical rules in different forms are rectified to facilitate the compilation and analysis. The rules for predicate p are *rectified* if all the functions are mapped to the corresponding functional predicates by the function-predicate transformation, and all the heads of the rules are identical and of the form p(X_1, ..., X_k) for distinct variables X_1, ..., X_k.

After rectification, the "append" cluster becomes :

append(U, V, W) :- U=[], V=W. (r_1'')

append(U, V, W) :- append(U_1, V, W_1), cons(X_1, U_1, U), cons(X_1, W_1, W). (r_2'')

Now, using the compilation technique discussed in Chapter 3, the recursion "append" can be compiled into the following chain form:

$$\text{append}(U_0, V, W_0) = \bigcup_{i=0 \text{ to } \infty} \left((\text{cons}(X_i, U_{i+1}, U_i), \text{cons}(X_i, W_{i+1}, W_i))^i, U_i = [], W_i = V \right).$$

4.2. Compilation of Nested Functional Linear Recursions

We further generalize our domain of discussion to **functional nested linear recursions**:

Definition [HaLu92].

A rule is **linearly recursive** if its body contains exactly one recursive predicate, and that predicate is defined at *the same deduction level* as that of the head predicate. A rule is **nested linearly recursive** if its body contains more than one recursive predicate but there is exactly one defined at the same deduction level as that of the head predicate. A recursion is **linear** if it consists of one linearly recursive rule and one or more non-recursive *exit rules*. A recursion is **nested linear** if every recursive predicate in the recursion is defined by one nested linearly or linearly recursive rule and one or more non-ecursive rules. A recursion is **function-free** if it does not contain function symbols; otherwise, it is **functional**.

Example 4-2. The n-queens problem:

(q_1) nqueens(N, Qs) :- range(1, N, Ns), queens(Ns, [], Qs).

(q_2) range(M, N, [M | N_s]) :- M < N, M_1 is M+1, range(M_1, N, N_s).
(q_3) range(N, N, [N]).

(q_4) queens(Unplaced, safe, Q_s) :- Select(Q, Unplaced, Unplaced_1),
not attack(Q, Safe),
queens(Unplaced_1, [Q|Safe], Q_s).
(q_5) queens([], Q_s, Q_s).

(q_6) attack(X, X_s) :- attk(X, 1, X_s).

(q_7) attk(X, N, [Y | Y_s]) :- X is Y+N.
(q_8) attk(X, N, [Y | Y_s]) :- X is Y–N.
(q_9) attk(X, N, [Y | Y_s]) :- N_1 is N+1, attk(X, N_1, Y_s).

(q_{10}) select(X, [X | X_s], X_s).
(q_{11}) select(X, [Y | Y_s], [Y | Z_s]) :- select(X, Y_s, Z_s).

The n-queens problem is a functional nested linear recursion because the IDB predicate "nqueens" is defined by one linear recursion "range" and one nested linear recursion "queens", while both recursions contain function symbols. The compilation of such a recursion is performed by first transforming it into a function-free recursion using the *function-predicate transformation* [Han92], and then compiling the recursions at every level by treating every lower level IDB predicates as EDB predicates [HaLu92].

The n-queens program is normalized as follows. Notice that cons(M, Ns, MNs) is a built-in list construction predicate indicating MNs = [M | Ns].

```
nqueens(N, Qs )  :- range(1, N, Ns ), queens( Ns , [], Qs ).

range(M, N, MNs)  :- M<N, M1 = M+1, cons(M, Ns, MNs), range(M1, N, Ns).
range(M, N, MNs)  :- M = N, cons(N, [], MNs).

queens(U, S, Qs )  :-  select(Q, U, U1),  not attack(Q, S), cons(Q, S, S1 ),
                       queens(U1, S1, Qs ).
queens(U, S, Qs )  :-  U = [], S = Qs.

attack(X, Xs) :-    attk (X, 1, Xs ).

attk(X, N, YYs) :-  X = Y + N, cons(Y, Ys, YYs).
attk(X, N, YYs) :-  X = Y - N, cons(Y, Ys, YYs).
attk(X, N, YYs) :-  cons(Y, Ys , YYs), N1 = N + 1, attk (X,  N1, Ys).

select(X, YYs, YZs) :-  cons(X, YZs, YYs).
select(X, YYs, YZs) :-  cons(Y, Ys, YYs), cons(Y, Zs, YZs),  select(X, Ys , Zs).
```

By normalization, the chain-predicate(s) of each recursion and their variable connections are made explicit :

The recursion **range(M, N, MNs)** is a *single chain recursion.* The variables M and MNs in the head predicate are connected to the corresponding variables M_1 and N_s respectively in the recursive predicate in the body via the chain predicates " $M < N$, $M_1 = M + 1$, cons(M, Ns, MNs)". The variable N is an *exit variable* (the variable in the corresponding argument positions of the recursive predicates in both the head and body of the rule) [Han89b].

Similarly, **queens(U, S, Qs)** is a *single chain recursion* where Qs is an exit variable, while U and S in the head are chain variables connected to U_1 and S_1 in the recursive predicate in the body through the chain predicates "select(Q, U, U_1), not attack(Q, S), cons(Q, S, S_1)".

Also, **select(X, YYs, YZs)** is a *single chain recursion* with X as an exit variable, while YYs and YZs in the head as chain variables connected to Ys and Zs in the body via the chain predicates "cons(Y, Ys, YYs), cons(Y, Zs, YZs)" .

Finally, **attk(X, N, YYs)** is a *double chain recursion* with X as an exit variable, while N and YYs in the head are chain variables connected to N_1 and Ys in the body via the chains "cons(Y, Ys, YYs)" and "$N_1 = N + 1$". □

4.3. Query Analysis on Compiled Functional Nested Linear Recursions

For functional recursions, we have the extra phase of **query analysis** between the compilation and evaluation phases. The reason is that without the consideration of functions, we can safely assume all the base relations involved are finite. Thus, using the semi-naive evaluation method, the intermediate relations generated at each iteration are finite and the evaluation will terminate in finite steps. However, with functions being considered as part of the base relations, we are facing the possibility of having infinite intermediate relations or non-terminable evaluations.

The compilation of a functional linear recursion greatly facilitates its analysis. Since many functional predicates are built-in predicates(e.g. arithmetic, relational and list functions) defined on infinite domains, the analysis must ensure that query evaluation can generate all the result set and terminate properly. Two major issues must be examined in the analysis phase: (1) **finite evaluability**, that is, the evaluation is performed on finite relations and generates finite intermediate relations at each iteration, and (2) **terminability**, that is, the evaluation generates all the answers and terminates in finite number of iterations.

Other enhancements, such as the detailed analysis of data types, can improve the efficiency of evaluation[HaLu92]. However, such enhancements will not affect the evaluability of a query and are therefore not discussed in this thesis.

4.3.1 Finite Evaluability of Functional Recursions

The justification of finite evaluability relies on both *query information* and *finiteness constraints*. A **finiteness constraint**, $X \rightarrow Y$, over a predicate r implies that each value of attribute X corresponds to a finite set of Y values in r [Han92]. Finiteness constraint is strictly weaker than the functional dependency studied in database theory [Ullm89a]. It holds trivially true for all finite predicates. Since all the EDB relations are finite, all the arguments in EDB relations satisfy the finiteness constraint. In a functional predicate $f(X_1, ..., X_n, V)$, if all the domains for arguments $X_1, ..., X_n$ are finite, V must be finite no matter whether f is a single or multiple valued function, that is, $(X_1, ..., X_n) \rightarrow V$.

Specific finiteness constraints should be explored for specific functions. In many cases, one argument of a function can be computed from the values of the other arguments and the value of the function. For example, in the functional predicate $+(X, Y, Z)$ (i.e. "$X+Y = Z$"), any argument can be finitely computed if the other two arguments are finite. Such a relationship can be represented by a set of *finiteness constraints*, such as $(X, Z) \rightarrow Y$, and $(Y, Z) \rightarrow X$. An important finiteness constraint, $Z \rightarrow (X,Y)$, holds in the functional predicate cons(X, Y, Z), which indicates if the list Z is finite, there is only a finite number of choices of X and Y [Han92].

Since query constants may bind some infinite domains of variables to finite ones, the analysis of finite evaluability should incorporate query instantiation information. Similar to the notations used in the Magic Sets transformation [BaRa86, Ullm89a], a superscript b or f is used to adorn a variable to indicate the variable being *bounded* (finite) or *free* (infinite), and a string of b's and f's is used to adorn a predicate to indicate the bindings of all its arguments.

Algorithm 4-1. Testing the finite evaluability of a query in an n-chain recursion [Han92].

Input.

(1) An n-chain recursion consisting of an n-chain recursive rule and a set of exit rules, (2) A set of finiteness constraints, and (3) query instantiation information.

Output. An assertion of whether the query is finitely evaluable.

Method.

1. Initialization: A variable is finite if it is in an EDB predicate or is equivalent to one or a set of constants.
2. Test the finite evaluability of (1) the exit rule set, and (2) the first expanded exit rule set (the rule set obtained by unifying the n-chain recursive rule with the exit rule set). This is done by pushing the query binding information into the rules being tested and propagating the finiteness bindings iteratively based on the following two *finiteness propagation rules*: (i) If there is a finiteness constraint $(X_1, \ldots, X_n) \rightarrow Y$ and X_i^b (for $1 \leq i \leq n$), then Y^b; and (ii) if ($X = Y$ or $Y = X$) and X^b, then Y^b.
3. Return *yes* if every variable in the two sets of rules being tested is finite after the finiteness binding propagation or *no* otherwise. □

Theorem 4-1. Algorithm 4-1 correctly tests the finite evaluability of an n-chain recursion in O(k) time, where k is the number of predicates in the recursion [HaWa91].

Example 4-3. For the recursion "append" defined in Example 4-1, we have $2^3 = 8$ possible query binding patterns for the head predicate append(U, V, W), the adornments are: bbb, bbf, bfb, bff, fbb, fbf, ffb and fff.

To analyze the finite evaluability, we consider the exit rule and the first expanded exit rule:

append(U, V, W) :- U = [], V = W. (e_0)

append(U, V, W) :- cons(X, U_1, U), cons(X, W_1, W), U_1 = [], W_1 = V. (e_1)

We go through the analysis of two cases as examples :

Case 1. Queries with the binding pattern ffb, such as " ?- append(U, V, [a, b, c])."

For e_0, the initial set of finite variables is {U, W}, U is finite because it equals to the constant "[]", W is finite because it is instantiated in the query. In the propagation of the binding information, V is also finite because V = W and W^b.

For E_1, the initial set of finite variables is {W, U_1}. The finiteness is then propagated as follows: (1) X and W_1 are finite because W^b and there is the finite constraint "$W \rightarrow (X, W_1)$" because "cons(X, W_1, W)"; (2) V is finite because ${W_1}^b$ and V = W_1; (3) finally, U is finite because of the constraint "$(X, U_1) \rightarrow U$" in "cons(X, U_1, U)".

We have proved that all the variables in e_0 and e_1 are finite after binding propagation, and thus query $append^{ffb}$ is finitely evaluable. Notice that the *order* of finiteness propagation is very important in deciding the evaluation order of the predicate, as we will see in the query evaluation phase [4.4].

Case 2. Queries with the binding pattern fbf, such as " ?- append(U, [1, 2], W)."

For e_0, we have the initial finite variable set {U, V}, and then W is finite because W=V and V^b.

For e_1, the initial finite variable set is {V, U_1}, we can have the binding information propagated to $W_1{}^b$ because V^b and V = W_1. However, the finite set {V, U_1, W_1) can not be extended any further with the predicates "cons(X_1, U_1, U)" and "cons(X_1, W_1, W)". Thus, the query is not finitely evaluable.

Notice that once we come to the conclusion that the query is not finite evaluable, the analysis is terminated without actually getting into the evaluation phase, and thus eliminates the possibility of getting into infinite loops. This is one of the valuable aspects of this evaluation method.

Following the same process, it can be easily proved that among the eight possible binding patterns of queries on append, only three cases : bff, fbf and fff are not finitely evaluable [Han92]. □

4.3.2. Termination of Query Evaluation

The second important part of query analysis on functional recursions is its **terminability**. Finite evaluability guarantees that intermediate relations at each iteration are finite, but this does not mean that the iterative processing will terminate in finite steps. To analyze the terminablity of the processing of a recursive query, we need to consider the "monotonicity constraints".

Definition.

A **monotonicity constraint** is a relationship $r_i \rangle r_j$, (" $\rangle$ " represents a partial order), where r_i and r_j are two arguments. The constraint holds for r_i and r_j if and only if r_i is strictly greater than r_j according to the partial order " $\rangle$ ".

Monotonic behaviour is typical in arithmetic functions, list functions and relational functions. Such behaviour should be used in query analysis and evaluation to determine the termination of the iteration.

Some examples of monotonic constraints are[Han92]:

1. An *acyclic EDB relation* is a partially ordered relation, and can be considered as a monotonicity constraint.
2. An *arithmetic operation* often implies the monotonicity of a function. For example, $F_1 + F_2 = F_3$ implies that $F_3 > F_1$ and $F_3 > F_2$.
3. The *monotonicity of a list function* is usually associated with the growing and shrinking of its length. For example, cons(X, L_1, L_2) implies that length(L_2) > length(L_1).

The process of evaluating a compiled linear recursion is essentially the iterative processing of a growing sequence of chain elements. An argument in the chain predicate is monotonic if its value has certain monotonic behaviour as the number of iterations increases. The query evaluation process terminates if there exists a **termination restraint** which blocks the growing or shrinking of the monotonic argument. A termination restraint can be provided by a query, an EDB relation or the natural constraints provided by the semantics of an argument, such as : " for any list L, length(L) > 0".

Notice that, since each chain has two ends, the **far end** (which shares variables with the query) and the **near end** (which shares variables with the exit rules), it can be evaluated in two directions, i.e. from the far end to the near end, or vise visa. Thus, we need to first decide which direction to proceed for each chain. The main factors need to be considered are : (i) which end is more selective and (ii) whether the processing can terminate properly following this direction. These analysis are done before the evaluation part. Actually we are generating an "execution plan" for every query before evaluating it. This again shows the systematic approach of this method.

Example 4-4. The queries on "append" are terminable if it is evaluable.

We study one simple case here, the general algorithm will be discussed in [4.4.1] when we consider *the chain-splitting evaluation* of functional recursions.

Consider the query pattern append^{bfb}, representing queries such as "?- append([a,b],V, [a,c,d])." . We will see that the evaluation should follow the chain "cons2(U_i, W_i, U_{i+1}, W_{i+1}) :- cons(X_i, U_{i+1}, U_i), cons(X_1, W_{i+1}, W_i)." from the far end to the near end, i.e. from (U_i, W_i) to (U_{i+1}, W_{i+1}) .

Because of the monotonic constraint that " if cons(X, L_1, L_2), then length(L_1) < length(L_2)", we have "length(U_{i+1}) < length(U_i)" and "length(W_{i+1}) < length(W_i)", which means that the length of the lists U_i and W_i shrinks when the iteration number i increases when proceeds from the far end to the near end. Moreover, at the other end of the chain (the near end in this case), we have the *termination restraint* that "length(U_{i+1}) = 0" because "U_{i+1} = []" , which would block the shrinking of U_i. Thus, the evaluation is terminable. □

This example shows that the terminability is associated with the following aspects: (1) the processing direction of the chains; (2) the monotonic behaviour of the arguments in the *driving chains* (chains evaluated in the forward process of evaluation in the case of multiple chains); and (3) the termination restraint on the terminating end of the driving chains.

4.4. Generating the Evaluation Plan for Functional Linear Recursions

In this section, we examine the process of generating the evaluation plan for a functional linear recursion. Before getting into the details, here is a diagram of the complete process:

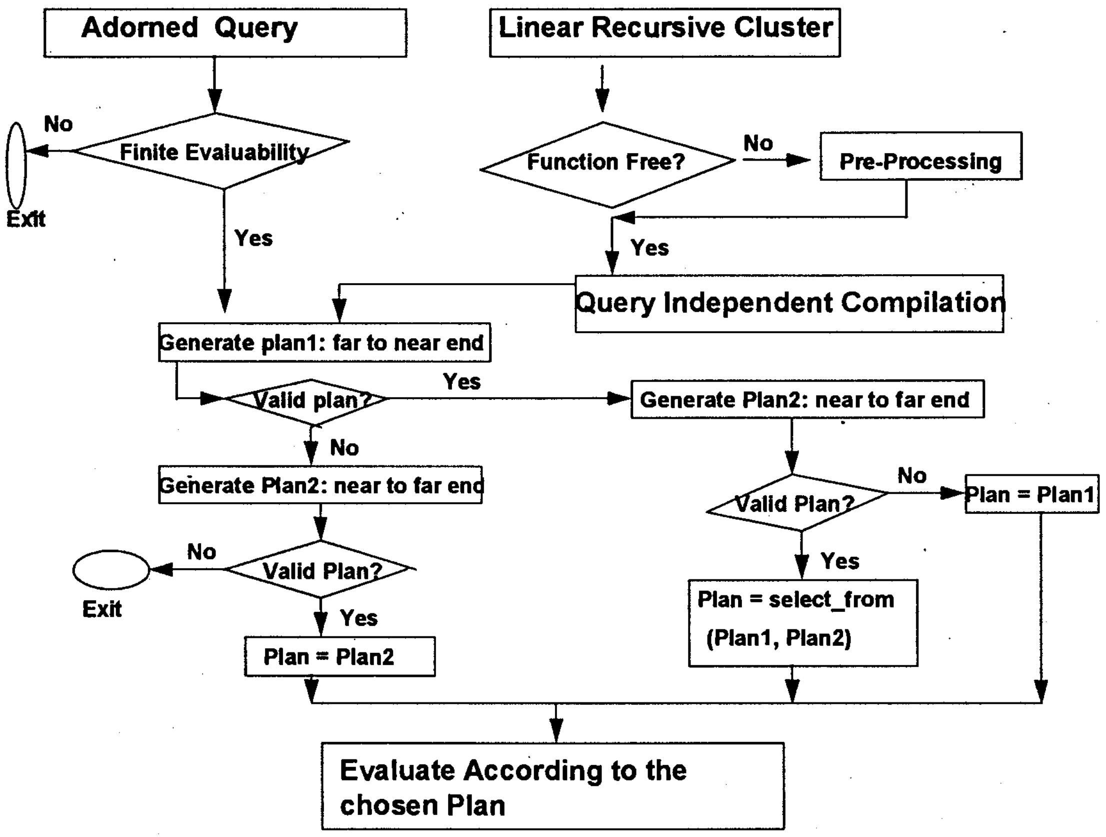

Adorned Query
Linear Recursive Cluster
Finite Evaluability
No
Exit
Yes
Function Free?
No
Pre-Processing
Yes
Query Independent Compilation
Generate plan1: far to near end
Valid plan?
Yes
Generate Plan2: near to far end
No
Generate Plan2: near to far end
Valid Plan?
No
Plan = Plan1
No
Exit
Valid Plan?
Yes
Yes
Plan = select_from (Plan1, Plan2)
Plan = Plan2
Evaluate According to the chosen Plan

Figure4-1. Diagram of Complete Chain Based Query Evaluation Method

The goal of this section is to generate an efficient query plan based on the input of (i) the compiled chain form of a recursion, and (ii) the query binding information. Three major issues will be discussed : (1) chain-following vs. chain-splitting; (2) existence checking vs. query closure; and (3) constraint pushing techniques.

4.4.1. Chain-following Evaluation vs. Chain-splitting Evaluation

A function-free n-chain recursion can be processed in three processing direction combinations[Han89b] :

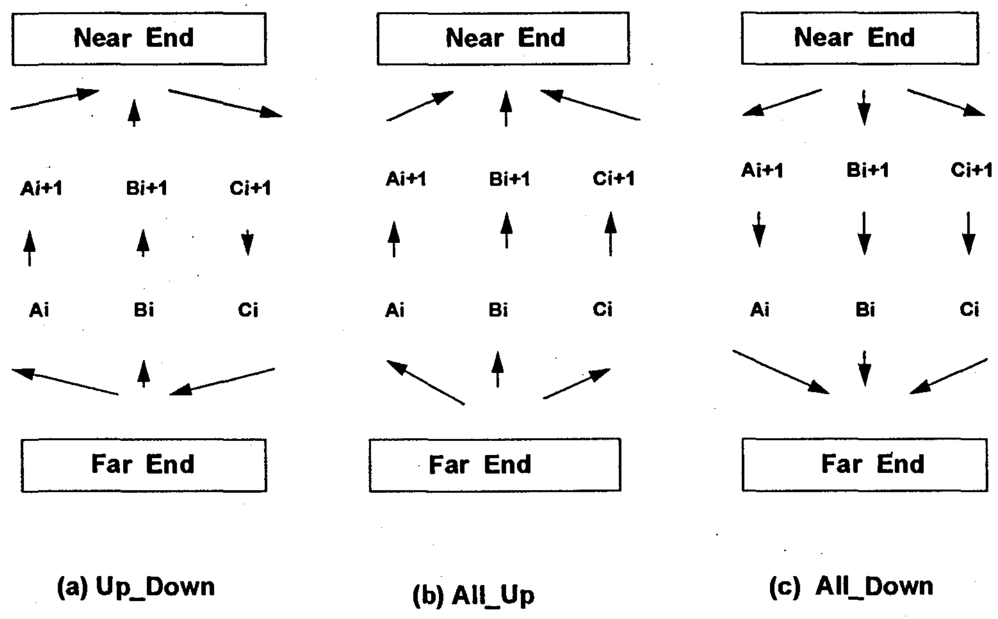


Figure4-2. Multi-Way Counting Method

1. **Up-Down**: start at the far end of some chains (the driving chains), climb up to the center, unify with the exit rules, and then step down along the remaining chains (the driven chains). In this case, the far end (or query end) is the "starting end" of the evaluation.

2. **All-Up**: start from the far end and synchronously climb up to the near end along all the chains. Obviously, the far end is the "starting end" and all the chains are "driving chains".

3. **All-Down**: start from the near end and synchronously step down to the near end along all the chains. In this case, the near end is the "starting end" and again all the chains are "driving chains".

There are many factors which influence the selection of processing directions, e.g. the selectivity of query constants, the size of the EDB predicates, the join selectivity of chain elements. Among these, the *selectivity* of the two ends plays a major role in quantitative analysis. For example, the Up-Down strategy should be used if the query end provides highly selectivity at only some of the chains. The All-Up processing should be used if the query provides high selectivity on all the chains, while the All-Down is used when the exit rules provides high selectivity on all the chains.

This method is called **chain-following evaluation**, and is applicable to all function-free n-chain recursions in a way similar to a generalized counting method [Han89b].

The chain-following evaluation method is also applicable to functional recursions if every predicate in a chain evaluation path is immediately evaluable. However, depending on the recursions and the available query bindings, some functional predicates in a chain generating path may not be immediately evaluable.

In this case, a driving chain may need to be partitioned into two portions: an *immediately evaluable portion* and a *buffered portion*[Han92]. The immediately evaluable portion is evaluated from the "starting end" of evaluation, while the evaluation of the buffered portion is delayed until the other end is reached and sufficient binding information is obtained. (Notice that the recursion must be *finitely evaluable* in the first place.)

The evaluation of the buffered portion of a driving chain is very similar to that of the driven chains, except that the shared variables between the immediately evaluable portion and the buffered portion of a chain must be "buffered" in the forward process and combined into the computation in the backward process.

Such evaluation technique is called the **chain-split evaluation** [HaWa91].

Example 4-5. Case study of the evaluation of "append".

From the compilation, we know that the recursion "append" is a single chain recursion. The chain is defined as "cons2(U_i, W_i, U_{i+1}, W_{i+1}) :- cons(X, U_{i+1}, U_i), cons(X, W_{i+1}, W_i).". The "*head*" of the chain cons2 is "cons2(U_i, W_i)" while the "*tail*" of the chain cons2 is "cons2(U_{i+1}, W_{i+1})"; the *far end* is < U_0, W_0 > and the *near end* is < U_n, W_n >. V is an "*exit variable*" which is in both the far end and the near end.

We study the evaluation plan of two queries against append:

Case1. Queries with the adorned predicate append^{bfb}(U_0, V, W_0), such as "?- append([a, b], V, [a, b, c])."

The far end is "append^{bfb}(U_0, V, W_0)", thus, the head of cons2 is "cons2^{bb}(U_i, W_i)". The near end is "append^{bff}(U_n, V, W_n)" (U_n is bounded because "append(U_n, V, W_n) :- U_n = [], V = W_n." and "[]" is a constant), and thus the tail of cons2 is "cons2^{bf}(U_{i+1}, W_{i+1})".

Since this is a single chain recursion, it should be processed either all-up or all-down:

All-up: Starts from the chain-head, considering the iteration i: we have the chain-head cons2^{bb}(U_i, W_i). Propagating the binding to the body of the chain, we get cons^{ffb}(X, U_{i+1}, U_i), cons^{bfb}(X, W_{i+1}, W_i). According to the finite evaluability constraints of "cons", both functional predicates are immediately evaluable, which generates the tail of the chain $cons2^{bb}(U_{i+1}, W_{i+1})$. Notice that the "tail" of iteration i is the "head" of iteration i+1, and the variable X is "internal" to the current iteration which does not need to be "remembered" in further iterations.

The analysis shows that all-up evaluation of cons2 is feasible in that it generates finite intermediate relations in each iteration. Next, we must check the terminability of the evaluation. Notice that in the all-up process, we have the monotonicity constraints : "$length(U_{i+1}) < length(U_i)$" because U_i and U_{i+1} satisfy the predicate "$cons(X, U_{i+1}, U_i)$". By checking the other end of the chain, we have the constraint "$U_n = []$", which means "$length(U_n) = 0$". Combine these two aspects, we have the *termination restraint*: "$length(U_{i+1}) = 0$" and the iteration terminates when the restraint is satisfied.

So far we have proved that $append^{bfb}(U_0, V, W_0)$ is (1)finitely evaluable and (2)terminable if evaluated all-up. Thus all-up is a valid direction of evaluation and an evaluation plan is generated that a chain following strategy should be used.

All-down: Similarly, we analyze the all-down evaluation process. Starting from the near end, we have the chain tail as "$cons2^{bf}(U_{i+1}, W_{i+1})$". Propagating the constraints into the chain body, we have $cons^{fbf}(X, U_{i+1}, U_i)$, $cons^{fff}(X, W_{i+1}, W_i)$, none of the two function predicates are finitely evaluable. Changing the order does not help either. Obviously, this is not a valid evaluation direction.

The conclusion is, to evaluate $append^{bfb}(U_0, V, W_0)$, we should follow the chain "$cons2(U_i, W_i, U_{i+1}, W_{i+1})$:- $cons(X, U_{i+1}, U_i)$, $cons(X, W_{i+1}, W_i)$." bottom-up (from $<U_i, W_i>$ to $<U_{i+1}, W_{i+1}>$) using partial transitive closure algorithm .

The evaluation process of query "?- append([a, b], V, [a, b, c])." is shown as follows[Han92]:

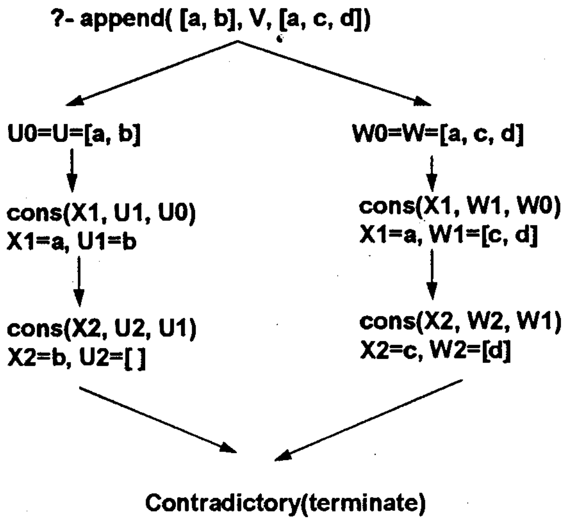


Figure 4-3. The evaluation of query " ?- append([a, b], V, [a, b, c])."

Case2. Queries with the adorned predicate $append^{ffb}(U_0, V, W_0)$, such as "?- append($U_0$, V, [a, b])."

The far end is "$append^{ffb}(U_0, V, W_0)$", so the head of cons2 is "$cons2^{fb}(U_i, W_i)$". The near end is "$append^{bff}(U_n, V, W_n)$" ($U_n$ is bounded because "append(U_n, V, W_n) :- U_n = [], V = W_n." and "[]" is a constant), and thus the tail of cons2 is "$cons2^{bf}(U_{i+1}, W_{i+1})$".

Using the same analysis method as in Case 1, we can see that following the chain either all-up or all-down will leave some or all function predicates on the evaluation path not immediately evaluable. However, the two directions are different: in the all-up evaluation, part of the chain is evaluable while in the all-down evaluation, none of the chain predicates are evaluable.

Thus, chain-splitting must be applied to the all-up evaluation as follows:

The chain "cons(X_i, U_{i+1}, U_i), cons(X_i, W_{i+1}, W_i)" should be partitioned into two portions: *the immediately evaluable portion* "$cons^{ffb}$(X_i, W_{i+1}, W_i)" and *the buffered portion* "cons(X_i, U_{i+1}, U_i)". The evaluation follows the immediate portion, which generates and buffers the value set < W_{i+1}, X_i > for each iteration i. When the termination condition (length(W_{i+1}) = 0) is satisfied, the far-end to near-end processing stops and the set of W_{i+1} generated in each iteration is unified with the exit rules which provides binding for V (W_{i+1} = V) and U_{i+1} (U_{i+1} = []). Next, the near-end to far-end processing starts which evaluates the buffered portion of the chain with the adornment "$cons^{bbf}$(X_i, U_{i+1}, U_i)". Notice that the values of X_i are generated and buffered during the first phase of process, while the bindings for U_{i+1} is obtained from the near end.

The evaluation process of "?- append(U, V, [a, b])." is shown as follows [Han92] :

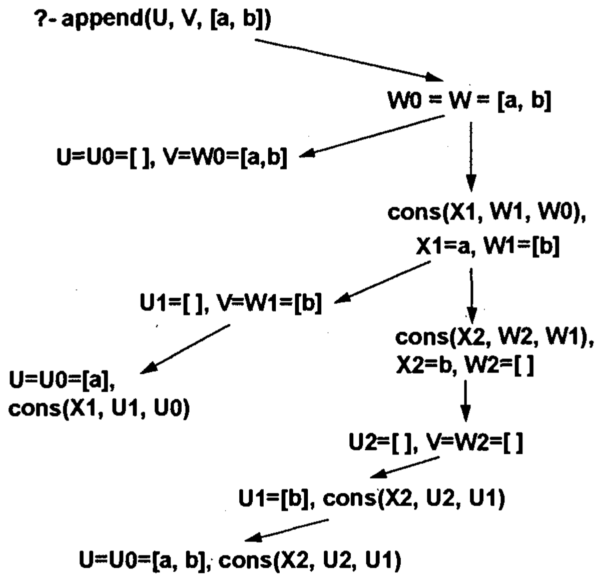


Figure 4-4. The evaluation of the query "?- (U, V, [a, b])." □

The above discussion is summarized into the following algorithm:

Algorithm 4-2. Chain-split evaluation of a *single-chain recursion* [Han92, modified] .

Input. A query and a compiled functional single-chain recursion.

Output. A query evaluation plan which applies the chain-split evaluation.

Method.

1. If every predicate in the chain generating path is immediately evaluable, the query can be evaluated without chain-split evaluation.

2. Otherwise, suppose (i) the chain is partitioned into two portions according to the available query bindings: an immediately evaluable portion p and a buffered portion q; (ii) p and q share the variable vector X_i on the i-th iteration; and (iii) the chain variables for p and q are U_i and W_i respectively, the evaluation proceeds in the following two phases:

```
(1) Phase 1 :
    i=0; $U_i$ = the initial set from the starting end;
        **For** iteration i, **do**
            **begin**
                If $U_i$ matches the termination restraint, k = i, break;
                    else **begin**
                        1) evaluate the immediate portion p based on $U_i$ :
                           compute the values $X_i$ and $U_{i+1}$;
                        2) buffer the set < i, $U_i$, $X_i$ >;
                        3) i ++; $U_i = U_{i+1}$;
                    **end**;
            **end**;
```

(2) Phase 2 :

```
For i = k to 0 do
  begin
    j = i ;
    while (j >0 ) do
      begin
        1) unify $U_j$ with the other end and compute the value $W_j$ ;
        2) compute the buffered portion based on $< X_j, W_j >$ and generate the
        value for $W_{j-1}$;
                  3) j = j-1; $W_j = W_{j-1}$;
      end;
    add $W_0$ to the result set;
  end;     □
```

4.4.2. Query Closure vs. Existence Checking

The evaluation methods we have discussed so far are based on the evaluation of **query closures**. However, in some cases, the computation of the entire query closure is not necessary.

As in Prolog, we differentiate two types of distinguished variables: an *inquired variable* (whose value is required by the query) and an *un-inquired* variable (whose value is not inquired by the query, denoted in Prolog by starting with an underscore, such as "_X"). For example, if we have a query "?- ancestor(a, _X).", we do not check the whole result set of a's ancestor, the computation should terminate (and return TRUE) if one of a's ancestor is found during the search. This type of computation is called **existence checking**.

In chain-based evaluation, there are two ends for each chain. In most cases, the evaluation should start at a more selective end and proceeds towards the other end of the chain. The evaluation computes the intermediate relations for each iteration using the *query closure* algorithm, and the computation stops when no more new results can be generated or some termination restraint is satisfied.

However, if the variables at the other end of the chain are not inquired, the computation could terminate whenever a value that satisfies the predicate is found. Applying existence checking in the chain-based evaluation method was first discussed in the evaluation of function-free linear recursiOns [Han89b], which is also applicable in the case of functional recursions. [Han92]

4.4.3. Constraint-Based Evaluation

The constraint based evaluation is another enhancement on the chain-based evaluation method. The idea is to combine query constraints into the evaluation in an earlier stage.

A query constraint usually adds constraint information to an IDB predicate. In a n-chain recursion, query constraints can be enforced at both ends of a compiled chain. It might be quite straightforward to push these constraints into the starting end of a chain, but with some special consideration, they can also be pushed into the terminating end of a chain, which usually helps to terminate the processing at an earlier stage. Detailed discussion is provided in [Han91a]

With some generalization, we have the following algorithm [Han92, modified] :

Algorithm 4-3. Chain-based query evaluation of a compiled functional single-chain recursion. [Fig4-1]

Input. A compiled functional single-chain recursion, a set of integrity constraints and a query.

Output. A chain-based query evaluation plan.

Method.

1. Test whether the query is finitely evaluable(Algorithm 4.1). If not, terminate.
2. Generate query evaluation plans :

```
for each of the two directions do
begin
   for each chain of the recursion do
      begin
         (1) partition the chain into two portions : the immediately evaluable portion
         and the buffered portion. (Notice that for EDB relations, the buffered portion
         is always empty.)

         (2) If the immediately portion is not empty: check the terminability of the
         evaluation on this portion. Basically this involves checking the monotonicity
         behaviour of the chain variables against the terminating end to form a
         termination restraint.

         If the evaluation is not terminable, break from the inner for loop. Otherwise,
         this is a valid driving chain on this direction.
      end; /* the inner for loop */

      if the set of driving chains is empty
            then this is not a valid evaluation direction
            else this is a valid evaluation direction, generate an evaluation plan.

end;  /* the outer for loop */
```

3. If there is at least one valid evaluation plan, evaluate the query:

   (1) If both directions are valid, choose the one that is more efficient based on the relative selectivity of the two ends.

   (2) If the finishing end is not inquired, apply the existence checking method, otherwise, apply the query closure method.

(3) Push the query constraints as early as possible according to the constraint-based evaluation algorithms. [Han91a]

(4) Return the result set.

Otherwise, print error message and terminate. (Notice that this eliminates the possibility of getting into infinite loops.) □

4.5. Evaluation of Nested Linear Recursions: the n-Queens Problem

Now that we have shown the advantage of using the chain-based evaluation method, that is, a systematic approach of evaluating functional linear recursions in a declarative and set-oriented way, we must have also noticed its weakness: the limited applicable domain of this method compared to general recursion handling systems, such as Prolog.

Our goal in this section is to push the applicable domain of the chain-based compilation and evaluation method further to functional nested linear recursions. Although the essential algorithms are mostly the same as the algorithms applied to single-layer recursions, this extension makes the method applicable to many more real life applications.

The basic idea of the extension is to process recursive predicates according to their "recursion level". Lower level recursions are treated as EDB predicates from the view of higher level predicates. A typical example : the n-queens program is used to show how the method works [HaLu92] .

The n-queens problem (Example 4-2) is a classical example used in the study of logic programming, constraint satisfaction and search methodologies. In [4.2], we have discussed how to compile nested linear recursions such as the n-queens problem into chain forms. In this section, we show the *analysis* and *evaluation* of this problem based on the compiled form and the query constraints.

Our study shows that the chain-based query analysis method generates efficient query evaluation plans not only for typical query bindings such as "?- nqueens(4, Qs).", but also for other kinds of queries, such as " ?- nqueens(N, [2, 4, 1, 3])" or "?- nqueens(N, [3, X, Y, 2])". Moreover, for queries that are not safe (not generating finite result sets), such as "?- nqueens(N, [2 | L])", the query processing terminates after the query analysis phase which detects the infiniteness and terminates by returning a warning message before getting into any infinite loops.

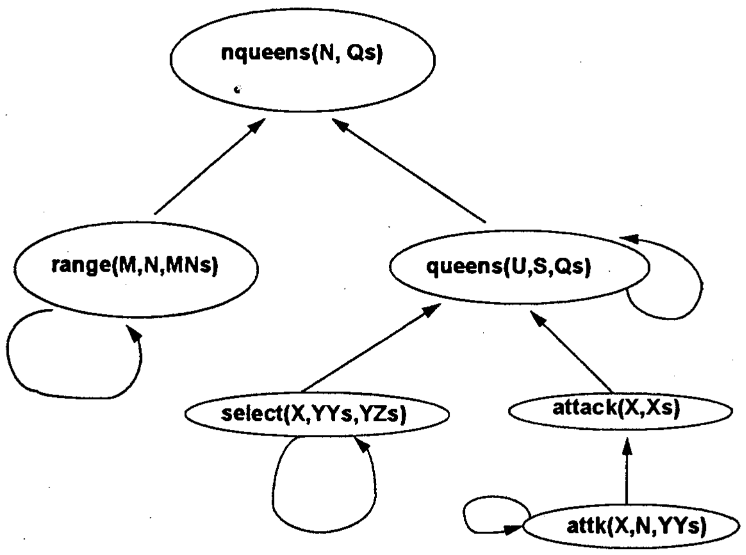


Figure 4-5. The recursive levels of the n-queens problem

Example 4-6. The analysis and evaluation of the n-queens problem[HaLu92].

Case 1. Queries with the adornment **nqueens^{bf}(N, Qs)**, such as "**?- nqueens(4, Qs)**."

Firstly, we propagate the query bindings and get evaluation plans for each predicate:

The binding "bf" for nqueens leads to the adorned rules: "nqueens^{bf}(N, Qs) :- range^{bbf}(1, N, Ns), queens^{bbf}(Ns, [], Qs)." or "nqueens^{bf}(N, Qs) :- queens^{fbf}(Ns, [], Qs), range^{bbb}(1, N, Ns)." However, analysis on the predicate "queens" shows that queens^{fbf} is not terminable in the evaluation of either direction, so we must evaluate the predicate "range" first. Notice that the analysis is done by the system and the order in which the rules are specified does not affect the evaluation process. This is the declarativeness that we want to achieve.

Using the same binding propagation technique, we have the adorned rule cluster as follows:

(q_1') $\text{nqueens}^{bf}(N, Q_s)$:- $\text{range}^{bbf}(1, N, N_s)$, $\text{queens}^{bbf}(N_s, [], Q_s)$.

(q_2') $\text{range}^{bbf}(M, N, MN_s)$:- $<^{bb}(M, N)$, $+^{fbb}(M, 1, M_1)$, $\text{range}^{bbf}(M_1, N, N_s)$, $\text{cons}^{bbf}(M, N_s, MN_s)$.

(q_3') $\text{range}^{bbf}(M, N, MN_s)$:- $=^{bb}(M, N)$, $\text{cons}^{bbf}(N, [], MN_s)$.

(q_4') $\text{queens}^{bbf}(U, S, Q_s)$:- $\text{select}^{fbf}(Q, U, U_1)$, not $\text{attack}^{bb}(Q, S)$, $\text{cons}^{bbf}(Q, S, S_1)$, $\text{queens}^{bbf}(U_1, S_1, Q_s)$.

(q_5') $\text{queens}^{bbf}(U, S, Q_s)$:- $=^{bb}(U, [])$, $=^{bf}(S, Q_s)$.

(q_6') $\text{select}^{fbf}(X, YY_s, YZ_s)$:- $\text{cons}^{ffb}(X, YZ_s, YY_s)$.

(q_7') $\text{select}^{fbf}(X, YY_s, YZ_s)$:- $\text{cons}^{ffb}(Y, Y_s, YY_s)$, $\text{select}^{fbf}(X, Y_s, Z_s)$, $\text{cons}^{bfb}(Y, Z_s, YZ_s)$.

(q_8') $\text{attack}^{bb}(X, X_s)$:- $\text{attk}^{bbb}(X, 1, X_s)$.

(q_9') $\text{attk}^{bbb}(X, N, YY_s)$:- $+^{bfb}(Y, N, X)$, $\text{cons}^{bfb}(Y, Y_s, YY_s)$.

(q_{10}') $\text{attk}^{bbb}(X, N, YY_s)$:- $-^{bfb}(Y, N, X)$, $\text{cons}^{bfb}(Y, Y_s, YY_s)$.

(q_{11}') $\text{attk}^{bbb}(X, N, YY_s)$:- $\text{cons}^{bfb}(Y, Y_s, YY_s)$, $+^{fbb}(N, 1, N_1)$, $\text{attk}^{bbb}(X, N_1, Y_s)$.

In order to compare the evaluation efficiency of different predicate ordering and adjust the predicate evaluation orders, more detailed analysis can be performed on the adorned program. For example, *cardinality analysis* can be performed on the program to determine the possible number of outputs for a particular set of query bindings. Obviously, card(M, N):card(MNs) = 1:1 in range^{bbf}(M, N, MNs). That is, for a particular pair of values of M and N, the recursion produces exactly one value for MNs. This can be deduced based on the properties of "cons". Moreover, the knowledge about the selectivity of bindings also benefits the analysis of processing efficiency. For example, the binding "U $=^{bb}$ []" is highly selective since there is only one possible value for U which satisfies the predicate "=" .

Finally, *termination analysis* must be performed on each compiled recursion based on the monotonicity behavior of the variables in the recursive predicates and the available binding information. For example, the recursion $range^{bbf}(M, N, MNs)$ terminates for the following reason: In the adorned recursive rule (q_2'), we have both "$M < N$" and "$M_1 = M+1$". The latter indicates "$M_1>M$", that is, the value in the first argument position monotonically increases; whereas the former indicates that the value in the first argument position must be less than the given value N. Thus, the iteration will terminate in finite steps. Similarly, we can show that the evaluation of other recursions with the given binding patterns can terminate as well.

The binding propagation analysis determines not only the predicate evaluation order but also the appropriate query evaluation strategies. For example, the adorned rules (q_1') to (q_{11}') indicate that chain-split evaluation should be performed on "$range^{bbf}(M, N, MNs)$" because the compiled chain "$M < N$, $M_1 = M + 1$, cons (M, Ns, MNs)" must be split into two "$M < N$, $M_1 = M + 1$" and "cons(M, Ns, MNs)" in the evaluation to guarantee finite evaluation (as shown by the predicate oredering in the body of (q_2')). Similarly, chain-split evaluation should be performed on "$select^{fbf}(X, YYs, YZs)$", chain-following evaluation on "$queens^{bbf}(U, S, Qs$)", and existence-checking evaluation on "$attk^{bbb}(X, N, YYs)$" .

Secondly, we examine in detail the evaluation process of the query "?- nqueens(4, Qs).", based on the adorned programs (q_1') to (q_{11}').

First, chain-split evaluation on "range(1, 4, MNs)" derives MNs=[1,2,3,4]. This proceeds as follows: (i) at the first iteration, range(1, 4, MNs) generates range(2, 4, Ns), cons(1, Ns, MNs), (ii) at the second iteration, range(2, 4, Ns) generates range(3, 4, Ns_2), cons(2, Ns_2, Ns), (iii) at the third iteration, range(3, 4, Ns_2) generates range(4, 4, Ns_3), cons(3, Ns_3, Ns_2), and finally, (iv) at the fourth iteration, range(4, 4, Ns_3) matches only the exit rule and produces Ns_3 = [4]. Thus, we have the result that MNs = [1, 2, 3, 4] .

The chain-following evaluation should be performed on the predicate "queens([1,2,3,4], [], Qs) ". The evaluation process is presented below iteration by iteration.

For the first iteration, select (Q, [1,2,3,4], U_1) derives the pairs of Q and U_1 values as: (Q, U_1): {(1, [2,3,4]), (2, [1,3,4]), (3, [1,2,4]), (4, [1,2,3])}. Every generated Q passes the test "not attack (Q, [])" and the iteration generates a table of inputs for the second iteration, queens(U_1, S_1, Qs), as: (U_1, S_1): {([2,3,4], [1]), ([1,3,4], [2]), ([1,2,4], [3]), ([1,2,3], [4])}. (Notice that the negation handling itself is a research area in deductive databases, which is out of the scope of discussion in this thesis.)

Similarly, the second iteration will generate a table of inputs for the third iteration. Take the first pair queens([2,3,4], [1], Qs) as an example. "Select (Q, [2,3,4], U_1)" derives the pairs of Q and U_1 values as: (Q, U_1): {(2, [3,4]), (3, [2,4]), (4, [2,3])}. (Notice that "Q=2" cannot pass the test "not attack (2, [1])"). It generates a table of inputs "queens(U_1, S_1, Qs)" for the third iteration as follows: (U_1, S_1): {([2,4], [1,3]), ([2,3], [1,4])}.

The process continues until the input set U for queens becomes empty, and the final result set to the query is Qs = {[2, 4, 1, 3], [3, 1, 4, 2]}.

The evaluation of the above query demonstrates that the compilation-based query evaluation has the following distinguished features in comparison with Prolog implementations: (i) appropriate predicate evaluation orders are determined automatically based on the compilation and the query binding analysis; (ii) efficient query evaluation strategies are selected and query evaluation plans are generated based on analysis of the predicate ordering and their associated bindings; and (iii) the processing generates all the answers to a query by set-oriented processing without backtracking [HaLu92].

Based on the principles discussed in the analysis of nqueens^{bf}(N, Qs), queries with other adornments, such as nqueens^{fb}(N, Qs), can be analyzed and evaluated similarly. Notice that the analysis and evaluation of complex logic programs with sophisticated query bindings, such as nqueens^{fb}(N, Qs), cannot be performed by Prolog implementations, nor by other previously studied deductive query evaluation techniques (according to the best of our knowledge).

Case 2. Queries with the adornment **nqueens^{fb}(N, Qs)**, such as "**?- nqueens(N, [2,4,1,3]).**"

In this case, the predicate queens should be evaluated first because it is unsafe to evaluate range^{bff}. Thus, the adorned program becomes,

(q_1") nqueens^{fb}(N, Q_s) :- queens^{fbb}(N_s, [], Q_s), range^{bfb}(1, N, N_s) .

(q_2") range^{bfb}(M, N, MN_s) :- cons^{bfb}(M, N_s, MN_s), $+^{fbb}$(M, 1, M_1),
range^{bfb}(M_1, N, N_s), $<^{bb}$(M, N).

(q_3") range^{bfb}(M, N, MN_s) :- $=^{bf}$(M, N), cons^{bbb}(N, [], MN_s).

(q_4") queens^{fbb}(U, S, Qs) :- queens^{ffb}(U_1, S_1, Qs), cons^{fbb}(Q, S, S_1),
not attack^{bb}(Q, S), select^{bfb}(Q, U, U_1).

(q_5") queens^{ffb}(U, S, Q_s) :- queens^{ffb}(U_1, S_1, Q_s), cons^{ffb}(Q, S, S_1),
not attack^{bb}(Q, S), select^{bfb}(Q, U, U_1), .

(q_6") queens^{ffb}(U, S, Q_s) :- $=^{fb}$(U, []), $=^{fb}$(S, Q_s).

(q_7") select^{bfb}(X, YY_s, YZ_s) :- cons^{bbf}(X, YZ_s, YY_s).

(q_8") select^{bfb}(X, YY_s, YZ_s) :- cons^{bfb}(Y, Z_s, YZ_s), select^{bfb}(X, Y_s, Z_s),
cons^{bbf}(Y, Y_s, YY_s).

(q_9") attack^{bb}(X, X_s) :- attk^{bbb}(X, 1, X_s).

(q_{10}") attk^{bbb}(X, N, YY_s) :- $+^{bfb}$(Y, N, X), cons^{bfb}(Y, Y_s, YY_s).

(q_{11}") attk^{bbb}(X, N, YY_s) :- $-^{bfb}$(Y, N, X), cons^{bfb}(Y, Y_s, YY_s).

(q_{12}") attk^{bbb}(X, N, YY_s) :- cons^{bfb}(Y, Y_s, YY_s), $+^{fbb}$(N, 1, N_1), attk^{bbb}(X, N_1, Y_s).

The binding propagation analysis is performed as follows.

The "fb" binding in nqueens(N, Qs) indicates that queens^{fbb}(Ns, [], Qs) must be evaluated first and then range^{bfb}(1, N, Ns); otherwise, the evaluation of range^{bff}(1, N, Ns) will lead to an unsafe (infinite) program.
Similarly, queens^{fbb}(Ns, [], Qs) indicates that the recursive predicate in the body queens(U_1, S_1, Qs) must be evaluated first. Otherwise, the evaluation will lead to an infinite program. The bindings in the head are propagated into the body, resulting in queens^{ffb}(U_1 , S_1, Qs) as shown in (q_4"). The further propagation in the recursive rule leads to the same binding pattern in the recursive predicate in the body, as shown in (q_5"). This evaluation order can be naturally viewed as evaluating first the exit rule portion and then the chain portion in the chain-based evaluation. In principle, a compiled chain can be evaluated in the order of either first-chain-then-exit or first-exit-then-chain. Since the former leads to an infinite program for the query whereas the latter leads to a finite one, the evaluation adopts the latter. The selection between the two orderings may also be based on the evaluation efficiency if both directions are feasible [HaLu92].

Binding propagation in the remaining program is similar to that in Case 1.

Next, we examine the evaluation of the adorned logic program for the query "?- nqueens(N, [2, 4, 1, 3])". First, the binding propagation leads to the evaluation of queens^{fbb}(Ns, [], [2, 4, 1, 3]), which should be evaluated in the first-exit-then-chain order according to our binding propagation analysis. The first iteration in the evaluation of the recursive rule proceeds as follows,

queens ([] , [2, 4, 1, 3], [2, 4, 1, 3]),

cons (Q, S, [2, 4, 1, 3]),

not attack (Q, S), select (Q, U, []).

This leads to Q = 2, S = [4, 1, 3] and U = [2]. The second iteration proceeds as follows,

queens ([2], [4, 1, 3], [2, 4, 1, 3]),

cons (Q, S, [4, 1, 3]),

not attack (Q, S), select (Q, U, []).

This leads to Q=4, S=[1,3] and U= {[2, 4], [4, 2]}.The third iteration proceeds as follows,

queens ({[2,4], [4, 2]}, [1, 3], [2, 4, 1, 3]),

cons (Q, S, [1, 3]),

not attack (Q, S), select (Q, U, []).

This leads to Q=1, S=[3] and U={[1,2,4], [1,4,2], [2,1,4], [2,4,1], [4,1,2], [4,2,1]}. Similarly for the fourth iteration. The result of the evaluation of queens(Ns, [], [2,4,1,3]) is Ns = {[1, 2, 3, 4], [1, 2, 4, 3], ..., [4, 3, 2, 1]}.

After the derivation of the set of Ns in (q_1"), the evaluation of range^{bfb}(1, N, Ns) proceeds in the order indicated in (q_2"), which derives N = 4, the only answer to the query.

Case 3. Queries with only uninstantiated or partially instantiated variables.

(1) Suppose that the binding pattern in nqueens(N,Qs) is "ff", such as "**?-nqueens(N, Qs)**". It is easy to verify that it is unsafe to evaluate the two predicates (no matter in what order), queens (Ns , [], Qs) and range (1, N, Ns), in (q_1"). Similarly, it is easy to show that the partially instantiated binding, "?- nqueens (N, [2|L])", will lead to the same problem. Although the partial binding "Qs = [2|L]" makes the initial evaluation proceed in a way similar to nqueens^{fb}, in the evaluation of the rule (q_4"), the evaluation of cons(Q, S, S_1) with the binding "S_1 = [2|L]" results in an uninstantiated (free) S, which leads to the same problem in the future evaluation.

Interestingly, if more detailed program behavior analysis were performed before query evaluation, certain queries containing only partially instantiated variables can be evaluated efficiently.

(2) We examine the query "?- nqueens (N, [3, X, Y, 2])" on the same compiled program. If the binding analysis similar to that in (1) were performed on the two predicates *queens* and *range* without further analysis, it would have concluded that the program is unsafe for query evaluation.

However, a more detailed program behavior analysis can be performed on the compiled program before query evaluation, which derives the list length relationships between the variables of type "list of integers" in the recursive predicates. In particular, we have: "length(Ns) = length(Qs)" in queens(Ns, [], Qs) and "length(Ns) = N" in range(1, N, Ns). The former is based on the property that "length(U) + length(S) = length(Qs)" in queens (U, S, Qs); while the latter is based on the property that "length(MNs)=N–M+1" in "range(M, N, MNs)". Notice that both properties are derived based on the basic property of list construction: "length (MNs) = length(Ns) + 1" in "cons(M, Ns, MNs)".

With the above two properties available, it is straightforward to derive N=4 from the fact that length ([3, X, Y, 2]) = 4. Thus, the query is equivalent to "?- nqueens (4, [3, X, Y, 2])" which can then be analyzed and evaluated in a way similar to Case 1. □

Based on the analysis of the n-queens problem, the process of query evaluation on (nested) linear recursions can be summarized into the following algorithm .

Algorithm 4-4. Query evaluation on (nested) linear recursions[HaLu92] .

Input: A linear or nested linear recursion and a query.

Output: The answer set to the query or a warning message if the query is not finitely evaluable.

Method:

1. Compile every linear recursion in the nested linear recursive cluster into a normalized recursion.
2. Perform type, cardinality or other behavior analysis on each normalized recursion.
3. Based on the available query bindings, perform *binding propagation analysis*. If the available bindings lead to an unsafe program, print a warning message and terminate the processing. Otherwise, proceed to the following steps.
4. Based on the result of binding propagation analysis, perform quantitative analysis on each normalized recursion, select efficient query evaluation strategy and generate a query evaluation plan.
5. Perform query evaluation and return the result set to the query. □

Chapter 5

The Conclusion

This chapter summarizes the research and implementation work of this thesis, and discusses the major advantages and disadvantages of this method compared to other recursion handling techniques.

5.1 Summary

This thesis discussed the implementation of the chain-based compilation method on linear recursions and the extension of the chain-based recursive query analysis and evaluation method to the domain of functional nested linear recursions.

Firstly, the implementation of the chain-based compilation method of linear recursions is discussed, which constructs a variable connection graph-matrix, the V-matrix, for a linear recursion and simulates recursion expansions by V-matrix expansions.

Based on the expansion regularity of linear recursions discovered by V-matrix expansions, compiled forms in either bounded or highly regular chain forms can be automatically generated for complex linear recursions.

The compilation of linear recursions into highly regular relational expressions facilitates the quantitative analysis of recursive queries and the generation of efficient query evaluation plans. Moreover, some bindings which are difficult to be captured by other techniques can be captured easily by the chain-based compilation method. Therefore, the automatic generation of compiled forms for linear recursions is a powerful tool for the analysis and evaluation of complicated linear recursions in deductive databases.

Secondly, the applicable domain of the chain-based compilation and evaluation method is extended to that of functional nested linear recursions.

A typical complex logic program containing nested linear recursions with list and arithmetic functions, the n-queens program, is analyzed using the chain-based compilation and evaluation method.

Our deductive query evaluation technique consists of query-independent compilation of logic programs and chain-based query analysis and evaluation of the compiled program. The method first compiles a logic program into a normalized program and then performs query binding propagation analysis, selects appropriate evaluation order, and derives efficient query evaluation plans for different query binding patterns. Some query bindings that cannot be evaluated by other logic programming techniques can be evaluated efficiently using the chain-based evaluation method.

The analysis of the n-queens program demonstrates that a large set of logic programs with linear or nested linear recursions can be evaluated efficiently using the compilation-based query analysis and evaluation technique. In comparison with other logic programming techniques, the deductive database approach shows its flexibility and efficiency on the analysis and evaluation of linear recursions.

5.2 Discussion

To conclude the thesis, we discuss the major advantages and disadvantages of the chain-based compilation and evaluation method compared to other recursion handling systems, and summarize the research work that has been done so far as well as the future research areas on this method.

1. In comparison with Prolog, the traditional logic programming language, it has the following two significant features, which are common for all deductive database approaches:

   - It has a pure declarative semantics. The advantage is that such methods handle optimization and termination systematically.

   - It uses a set-oriented data processing strategy. The advantage is that such methods are much more efficient when processing large amount of data stored in mass memory.

2. In comparison with some other deductive database query optimization methods, such as the magic sets method, it has the following advantages :

   - By query-independent compilation, our method generates highly regular chain forms which facilitates systematic query analysis and efficient query processing .

- By using the highly regular chain forms, this method is better in capturing query bindings and integrating constraints to optimize the query evaluation .
- By extending the applicable domain, this method is able to handle functional linear recursions while the magic sets method can only handle function free recursions .

The major limitation of this method is that its applicable domain is limited to the recursions compilable to chain forms, whereas Prolog can handle general recursions with nested levels of functions and the magic sets method is applicable to general function free recursions.

Major research works done so far on the chain-based compilation and evaluation method are [Han89a, Han89b, Han91a, Han92 and HaZe92]. More research is being conducted on extending the applicable domain of this method to non-linear recursions and incorporating some important features such as aggregation functions and negation handling into the query evaluation . Furthermore, it is important to perform further research into the issues on the extensions of the compilation and chain-based query evaluation method to deductive and object-oriented databases.

Reference

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

[BaRa86] F. Bancilhon and R. Ramakrishnan, "An Amateur's Introduction to Recursive Query Processing Strategies", Proc. 1986 ACM-SIGMOD Conf. Management of Data, pp.16-52, Washington, DC, May 1986.

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

[BrJa84] M. Brodie and M. Jarke, "On Integrating Logic Programming and Databases", Proc.1st Int. Workshop on Expert Database Systems, Kiawah Island, SC, October 1984.

[BrSa89] A. Brodsky and Y. Sagiv, "On Termination of Datalog Programs", Proc. 1st Int. Conf. Deductive and Object Oriented Databases (DOOD'89), pp.95-112, Kyoto, Japan, December 1989.

[CeGoLe89] Stefano Ceri, Georg Gottlob, and Letizia Tanca, "What You Always Wnated to Know About Datalog (And Never Dared to Ask)", IEEE Trans.on Knowledge and Data Engineering, Vol.1, No.1, pp.146-166, March1989 .

[LDL90] D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur and C. Zaniolo, "The LDL System Prototype", IEEE Trans. Knowledge and Data Engineering, pp.76-90, 2(1),1990.

[Dahl82] V. Dahl, "On Database Systems Development Through Logic", ACM Trans. Database Syst., 7(1), 1982.

[Date86] C. Date, "An Introduction to Database Systems", Addison-Wesley, 1986.

[GMN84] H. Gallaire, J. Minker and J. Nicolas, "Logic and Databases: A Deductive Approach", ACM Comput. Surv., 16(2), pp.153-185, 1984.

[GrMi92] J. Grant and J. Minker, "The Impact of Logic Programming on Databases", Comm. of ACM, 35(2), pp.66-81, 1992.

[HaLu89c] J. Han and W. Lu, "Asynchronous Chain Recursions", IEEE Trans. Knowledge and Data Engineering, 1(2), pp.185-195, 1989.

[HaLu92] J.Han and T.Lu, "N-Queen Problem Revisited: A Deductive Database Approach", Proc.1992 IJCSLP Workshop on Deductive Databases, pp.48-55, Washinton D.C., Nov. 1992.

[Han89a] J. Han, "Compiling General Linear Recursions by Variable Connection Graph Analysis", Computational Intelligence, 5(1), pp.12-31, 1989.

[Han89b] J. Han, "Multi-Way Counting Method", Information Systems, 14(3), pp.219-229, 1989.

[Han91a] J. Han, "Constraint-Based Reasoning in Deductive Databases", Proc. 7th Int. Conf. Data Engineering, pp.257-265, Kobe, Japan, April 1991.

[Han91b] J. Han, "Is It Possible to Capture More Bindings than Magic Rule Rewriting?", Proceedings of Int'l Workshop on Deductive Databases (in conjunction with 1991 Int'l Logic Programming Symp.), pp.10-19, San Diego, CA, October 1991.

[Han92] J. Han, "Compilation-Based List Processing in Deductive Databases", In A. Pirotte, C. Delobel, and G. Gottlob,(eds.), Extending Database Technology - EDBT'92 [Lecture Notes in Computer Science 580], pp.104-119, Springer-Verlag, 1992.

[HaWa91] J. Han and Q. Wang, "Evaluation of Functional Linear Recursions: A Compilation Aroach", Information Systems, 16(4), pp.463-469, 1991.

[HaZe92] J.Han and K.Zeng, "Automatic Generation of Compiled Forms for Linear Recursions", Information Systems, 17(4), pp.299-322, 1992.

[HaZeLu93] J.Han, K.Zeng and T.Lu "Normalization of Linear Recursions in Deductive Databases", Proc.9th Int'l Conf on Data Engineering, Vienna, Austria, pp559-567, April 1993.

[Hent89] P. van Hentenryck, "Constraint Satisfaction in Logic Programming", MIT Press,1989.

[Imie87] T.Imielinski, "Intelligent Query Answering in Rule Based Systems", J.Logic Programming, 4, pp.229-257, 1987.

[JaLa87] J. Jaffar and J-L. Lassez, "Constraint Logic Programming", Proc. 14th ACM Symp. Principles of Programming Languages, pp.111-119, Munich, Germany,1987.

[Jian90] B. Jiang, "A Suitable Algorithm for Computing Partial Transitive Closures", Proc. 6th Int. Conf. Data Engineering, pp.264-271, Los Angeles, CA, February 1990.

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

[KRBM89] D. B. Kemp, K. Ramamohanarao ,I. Balbin and K. Meenakshi, "Propagating Constraints in Recursive Deductive Databases", Proc. 1989 North American Conf. Logic Programming, pp.981-998, Cleveland, OH,October 1989.

[La90] J.L. Lassez, "Query Constraints", Proc. 9th ACM SIGACT-SIGMOD-SIGART symp on PODS, April 1990, Nashville, Tennessee, pp.288-198.

[Mink88] J. Minker, "Foundations of Deductive Databases and Logic Programming", Morgan Kaufmann, 1988.

[NeSt89] E. Neuhold and M. Stonebraker, "Future Directions in DBMS Research (Laguna Beach Report)",ACM SIGMOD Record, 18(1), 17-26, 1989.

[RBS87] R. Ramakrishnan, F. Bancilhon and A. Silberschatz, "Safety of Recursive Horn Clauses with Infinite Relations", Proc. 6th ACM Symp. Principles of Database Systems, pp.328-339, San Diego, CA, March 1987.

[SaVa89] Y. Sagiv and M. Vardi, "Safety of Datalog Queries over Infinite Databases", Proc. 8th ACM Symp. Principles of Database Systems, pp.160-171, Philadelphia, PA,March 1989.

[Sagi90] Y. Sagiv, "Is There Anything Better than Magic?", Proc. 1990 North American Conf. Logic Programming, pp.236-254, Austin, Texas, October 1990.

[ShKe84] D.E. Shepherd and L. Kerschberg, "Constraint Management in Expert Database Systems", Proc. 1st Int. Workshop on Expert Database Systems, Kiawah Island, SC,October 1984.

[SSU91] A. Silberschatz, M. Stonebraker and J. D. Ullman, "Database Systems: Achievements and Opportunities",Comm. ACM, 34(10), pp.94-109, 1991.

[StSh86] L. Sterling and E. Shapiro, "The Art of Prolog", The MIT Press, 1986.

[Ston88] M. Stonebraker, "Readings in Database Systems", Morgan Kaufmann,1988.

[Ston90] Michael Stonebraker, Lawrence A. Rowe and Michael Hirohama, "The Implementation of the POSTGRES", IEEE Trans. Knowledge and Data Engineering, 2(1), pp.125-141, 1990.

[TYZ89] D. Troy, C.T. Yu and W. Zhang, "Linearization of Nonlinear Recursive Rules", IEEE Trans. Software Engineering, 15(9), pp.1109-1119, 1989.

[Tsur91] S. Tsur, "Deductive Databases in Action", Proc. 10th ACM Symp. Principles of Database Systems, pp.142-153, Denver, CO, May 1991.

[Ullm89a] J. D. Ullman, "Principles of Database and Knowledge-Base Systems, Vols. 1 & 2", Computer Science Press, 1989.

[Ullm89b] J. D. Ullman, "Bottom-up Beats Top-down for Datalog", Proc. 8th ACM Symp. Principles of Database Systems, pp.140-149, Philadelphia, PA, March 1989.

[Ullm91] J. D. Ullman, "A Comparison of Deductive and Object-Oriented Database Systems", Deductive and Object-Oriented Databases (DOOD'91)[Lecture Notes in Computer Science 566], pp.263-277, Springer Verlag, 1991.

[UllZa90] J.D.Ullman and C.Zaniolo, "Deductive Databases: Achievements and Furture Directions", SIGMOD RECORD, Vol19, No.4, December 1990.

[YHH88] C. Youn, L. J. Henschen and J. Han, "Classification of Recursive Formulas in Deductive Databases", Proc. 1988 ACM-SIGMOD Conf. Management of Data, pp.320-328, Chicago, IL, June 1988.

[Zani84] C. Zaniolo, "Prolog : A Database Query Language For All Seasons", Proc. 1st Int. Workshop on Expert Database Systems, Kiawah Island, SC, October 1984.