Department of Computer Science
University of Cape Town

Semantic Optimisation in Datalog Programs

by Mark P.Wassell

A Thesis
Prepared Under the Supervision of
Associate Professor P.T.Wood
In Fulfilment of the Requirements for the
Degree of Master of Science in
Computer Science

University of Cape Town
December 1990

Abstract

Datalog is the fusion of Prolog and Database technologies aimed at producing an efficient, logic-based, declarative language for databases. This fusion takes the best of logic programming for the syntax of Datalog, and the best of database systems for the operational part of Datalog.

As is the case with all declarative languages, optimisation is necessary to improve the efficiency of programs. Semantic optimisation uses meta-knowledge describing the data in the database to optimise queries and rules, aiming to reduce the resources required to answer queries.

In this thesis, I analyse prior work that has been done on semantic optimisation and then propose an optimisation system for Datalog that includes optimisation of recursive programs and a semantic knowledge management module. A language, DatalogIC, which is an extension of Datalog that allows semantic knowledge to be expressed, has also been devised as an implementation vehicle. Finally, empirical results concerning the benefits of semantic optimisation are reported.

CR Categories H.2.0 [**Database Management**]: General; H.2.2 [**Database Management**]: Physical Design—*access methods*; H.2.4 [**Database Management**]: Systems—*query processing*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving— *logic programming*; *resolution*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods and Search—*heuristic methods*; *plan execution*; *formation*; *generation*

Keywords Deductive database, semantics, optimisation, Prolog, integrity constraints, residues, inference, recursive programs.

Acknowledgments

Many thanks to my supervisor, Associate Professor Peter Wood, for his assistance and guidance through the various stages of this work. I would also like to thank the Computer Science department here at UCT and wish them well for the future.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Datalog is a fusion of Prolog and database technologies aimed at producing a logic-based declarative language that is efficient and yet has the good features of logic programming languages. One benefit of this is the creation of expert system-like front-ends for relational databases. A Datalog program is similar to a Prolog one except that it has a simpler syntax and is evaluated differently.

Declarative languages have the property that the programmer only has to concentrate on what needs to be done and can ignore how it is done. One class of declarative languages are the logic programming ones. These languages have great expressive power, can model many different situations, and have a strong mathematical foundation. The best known logic programming language is Prolog, which has been implemented and studied extensively. However, it is neither logically complete nor sound and has some unsavoury procedural aspects. Furthermore, if you wished your Prolog program to process large amounts of data, it would show great inefficiency and you may be forced to revert to a procedural or a database language. This efficiency problem is solved by looking at database management systems and what is known about managing large databases. Section 2.1.1 continues with the reasons behind the development of Datalog.

Since the user of a declarative language is not required to know the procedural meaning of a program, the responsibility for efficient execution rests completely with the system and its designers. *Optimising* the program is one way that the system can improve efficiency. In this way optimisation of Datalog programs has become an important subfield. Of course, optimisation is not restricted to declarative languages and has appeared in the context of procedural languages. In addition, the definition of optimisation, as the rewriting of a

program so that its execution consumes less resources, applies to both types of language.

One uses the term *semantics* to refer to the meaning of a string of symbols. For example, C programs have a semantics, based on state transitions, built up out of the semantic meaning of each component, the simplest being the assignment statement. In simple relational databases the semantics relate the tuples in the database to the desired portion of the world being modeled by the database designer. For example, in a database we could map a relation called *parent* onto the concept of a parent used in family situations. It is also possible to attach more than one meaning to the data in the database as long as it is done consistently. In order to enforce a particular semantic meaning onto the database we use *integrity constraints*, such as functional dependencies, to restrict the allowable tuples. It is these that make up semantic knowledge and describe more about the world being modeled. Through semantic knowledge we have some idea of what data the Datalog program is going to be run against, so it may be possible to rewrite the program accordingly. When considering the equivalence of the original program with the optimised one, we only have to look at databases satisfying the set of constraints, instead of all possible databases. The importance of this is that our set of equivalent programs will be larger than if we were doing syntactic optimisation. This process has been called *semantic optimisation*.

In this thesis, I analyse some of the current ideas on semantic optimisation and then synthesize them into a homogeneous system, parts of which I have implemented.

Chapter 2 gives the background to the dissertation and draws on [GMN84, Ull85, BR86, Ull88, CGT89]. I describe the rationale behind the development of Datalog and then the Datalog language itself. Using the theory developed for logic programs, I give an introduction to the various possible semantic interpretations of Datalog programs, especially the fixed-point semantics of [Llo87]. Finally, I describe how we can evaluate Datalog programs, and then give an overview of optimisation.

Chapter 3 is the main part of the analysis phase of this thesis and here I describe semantic optimisation. First, I give some background to semantic optimisation. This includes some motivating examples, a short introduction to the theory, and a discussion on types of semantic knowledge and how older forms of semantic knowledge can be converted to Horn clause form semantic knowledge. I also give a survey of some semantic optimisation systems, from [Kin81, JCV84, Jar86]. In Section 3.2, I detail the semantic optimisation

system of [CGM87] (which I will label CGM). The key contribution of this system is its logic-based approach and the concept of residues, formed by merging rules and integrity constraints. In the next section, I introduce the semantic query optimisation system of [SO89] (which I label SO). Although designed for a relational database environment, it can be converted for use in a Datalog environment, a topic I detail in Section 4.5.2. I present both these systems within a common framework and fill in some gaps, such as completeness theorems, the main purpose being to dovetail these together to form the core of the system I have devised.

In Chapter 4, I describe semantic optimisation in DatalogIC, the language I have developed for semantic optimisation. This is the main synthesis part of the dissertation, and many of the ideas in this chapter have not been presented this way before. I first introduce DatalogIC, describe how it differs from Datalog and give its syntax. Then I give a sample session on the DatalogIC system. The rest of the chapter details the semantic optimisation system, which can be broken down into four parts: The first is the semantic knowledge manager, which is a knowledge-base holding the semantic knowledge entered in the DatalogIC program. Then we have the recursive program optimiser and the single rule optimiser. The latter is a synthesis of the the SO and CGM systems introduced in the previous chapter. Finally, we have the global optimiser which controls the optimisation process. Only the local optimiser has been implemented fully, while only the simplest of constraint managers and global optimisers has been implemented.

Chapter 5 deals with the implementation of the DatalogIC system, detailing those parts not mentioned elsewhere. The system was written in the C programming language, on the UNIX [1] operating system, using the Oracle [2] database management system as the backend. I give an overview of the system and then discuss the user-interface module and the evaluation algorithm used. The latter converts the program into SQL [3] statements and interacts with Oracle. Finally in this chapter, I present some empirical results which indicate that semantic optimisation can be beneficial.

Chapter 6 concludes the dissertation and mentions further work. Appendix A details the data structures and the algorithms used in the implementation, including those used

[1] UNIX is a registered trademark of AT&T.

[2] Oracle is a registered trademark of Oracle Corporation.

[3] Structured Query Language.

in the rule optimisation system.

Chapter 2

Datalog Programs and Their Evaluation

The aim of this chapter is to introduce the ideas covered in the rest of this dissertation and also to mention some related ideas. The first section gives a grounding in basic Datalog [1], the language used to write rules and queries, and outlines where it fits into the Prolog family of languages. Section 2.2 covers some of the theoretical aspects of Datalog, relating them to the theory of logic programming. Also introduced are some important concepts used when comparing Datalog programs and investigating their properties. The final two sections outline query evaluation and optimisation techniques. A good introduction, which covers most of what I mention below, is [BR86].

2.1 Datalog

Datalog is a recent development in the deductive databases field, being an offshoot of Prolog aimed at dropping several undesirable features of the latter while incorporating the good aspects of database management systems (DBMS). Logic programming languages have been considered desirable as front ends for DBMS [BJ86], but there are fundamental differences between databases and logic programming languages which have to be resolved. These differences arise due to the background of logic programming (LP) and databases: the former has strong theoretical foundations while the latter is more practically orien-

[1] The first use of the word "Datalog", to describe the language we are dealing with, was in [MW88], although ideas on a logic-based database language were considered prior to it

tated. I will first outline the reasons for the development of Datalog and then introduce the language itself, pointing out how it differs from Prolog. The semantics of the language are dealt with in more detail in Section 2.2.

2.1.1 The Rationale Behind Datalog

In this subsection, I consider how Datalog arises out of the intersection of the two fields of Prolog and database management systems. I first look at the good and bad aspects of Prolog and then do the same for database management systems. I follow closely the arguments given in [BJ86, Zan86].

The following features of Prolog have contributed to its use as a programming language:

- Logic programs are concise and can represent complex knowledge.
- There is a general inference mechanism and uniform representation scheme.
- It is a declarative language and thus the programmer need only specify what needs to be done and can ignore how it is to be done.
- Since logic programming is well understood, and a lot of research has been done on the theory of logic programming, the developers of LP systems are assured of a complete specification for the language.

Although Prolog is adequate for most deductive systems this is not so for deductive databases. These are characterised by having a large underlying database, which is usually relational and most of which will be on secondary storage. The reasons for the inadequacy of Prolog are:

- The top-down Prolog evaluation algorithm works with a tuple at a time. If one wanted, for instance, the set of bindings for a variable in a query, the Prolog interpreter would have to "fail" after each answer and backtrack to find the next binding. With large amounts of data this can be very inefficient.
- The Prolog operational semantics is not complete. There are times when the left to right, top-down, interpreter will loop, such as with the program:

$$\begin{aligned} ancestor(X,Y) &:- ancestor(X,Z), parent(Z,Y). \\ ancestor(X,Y) &:- parent(X,Y). \end{aligned}$$

and the query $ancestor(a, W)$.

- There is no system of types and objects as in other languages.
- The programming environment is generally poor and Prolog is difficult for novices to grasp.
- It lacks good support for updates, deletions and other features of database systems.

Thus a more data-orientated approach is required which will lead to efficient evaluation algorithms and greater accessibility for the user of the database.

Database management systems have come a long way since their inception and the following points, good and bad, have influenced their integration with logic programming:

- DBMSs have been designed with the problems of managing large amounts of data in mind, supporting many users and working with distributed data.
- Database languages lack the ability to express concisely complex relationships between data, such as recursive relationships.
- If the database language is procedural, the user has the problem of learning the insides of a DBMS, a time consuming task for the developer of applications. Even if the language is declarative, like SQL, the user may have be aware of how the system evaluates the statements in order to write efficient ones.

Thus Prolog is not suitable as a database language since it lacks many of the features commonly associated with database systems. However, as we have seen, there are some useful features that we would like to incorporate into database systems. The next task is to devise a language which pulls together the best of Prolog and the best of database management systems. This is the topic of the next subsection.

2.1.2 The Datalog Language

The syntax of a Datalog program is similar to that of Prolog except that the extra-logical features (such as "cut" and "assert") have been dropped and the language simplified so that linkage to the database backend is eased. I will first give the formal definition of a Datalog program and then discuss why some of the features are necessary [CGT89, CGT90, Ull88].

Definition 2.1.1 In what follows I will be using the usual definitions of first-order constructs, such as predicates, literals and Horn clauses. In summary these are as follows:

- Our alphabet of symbols is made up of
  - Words beginning with an uppercase letter for variables. (Letters from the last part of the alphabet, ie X, Y, Z, will be used as variables).
  - Words beginning with a lowercase letter for constants and predicate names. (Letters from the first part of the alphabet, ie a, b, c, will be used as constants, and letters from around the middle of the alphabet, i.e. p, q, r, will be used for predicate names).
  - Commas to denote conjunction, and $:-$ and $\rightarrow$ to denote implication.
- A *literal* is a string of the form $p(t_1, \ldots, t_n)$. p is the name of the predicate, which has *arity* n, and t_i is a *term* which is either a variable, a constant, or the "don't care" variable "_" (as used in Prolog).
- A *clause* is a disjunction of literals either negated or unnegated.
- A *ground* clause is a clause where all the terms are constants.
- A *Horn clause* is a clause with at most one unnegated literal. Often we write the Horn clause

  $$\neg A_1 \vee \ldots \vee \neg A_n \vee A$$

  either as $A :- A_1, \ldots A_n$ or as $A_1, \ldots, A_n \rightarrow A$. The list $A_1, \ldots, A_n$ is thus a conjunction of literals and is called the *body* of the clause. Note that we sometimes view a conjunction as a set of literals, and thus use $\subseteq$ to mean "sub-conjunction".
- A *fact clause* is a Horn clause with an empty body.

□

As we are dealing with "real world" programs we will often be using the comparison operators $\{>, \geq, <, \leq, =, \neq\}$. These are called *evaluable predicates* and have some useful properties which will be used throughout this thesis, especially in the SO system, detailed in Section 3.3. These properties are as follows:

1. All, except $\neq$, are transitive.

2. $\neq$ and $=$ are symmetric, and $=$, $\geq$ and $\leq$ are reflexive.

3. They are closed under negation. For instance, $\neg(a > b)$ is $a \leq b$. So we are prepared to accept negated evaluable predicates in the body of a Horn clause since these can be rewritten as unnegated predicates.

The relations corresponding to evaluable predicates are infinite, and so cannot be used without restriction in a database system. Thus they force us to impose several syntactic restrictions on the syntax of a Datalog rule, which I detail later.

Definition 2.1.2 A Datalog *rule* is a Horn clause which has the form

$$p(\overline{X}) \quad :- \quad q_1(\overline{X}_1), \ldots, q_n(\overline{X}_n), \delta$$

where:

1. $p(\overline{X})$ is the *head predicate* of the rule, each $q_i(\overline{X}_i)$ is a *predicate occurrence* and $q_1(\overline{X}_1), \ldots, q_n(\overline{X}_n), \delta$ is the *body* of the rule.

2. $\overline{X}_i$ are vectors of variables or constants (unlike Prolog no functions are allowed). These we call the *arguments* of q_i.

3. δ is a list of evaluable predicates of the form x_1 *op* x_2 where *op* is one of $>, \geq, <, \leq, \neq$ or $=$, and x_i is a variable or a constant.

We have to ensure that all variables in the head have a finite domain, so we say that they must be *limited* [Ull88]. This means that any variable appearing in the head of the rule must appear in a non-evaluable predicate in the body, or be connected by a chain of $=$ predicates to such a variable or to a constant. This is related to a rule being *range restricted* [CGM87] which states that each variable in the head must appear in at least one non-evaluable predicate. However, both restrictions limit the domain of the head variables, but the latter is unnecessarily restrictive. □

Definition 2.1.3 A *substitution* is a set of bindings for variables and is of the form

$$\{t_1/v_1, \ldots, t_n/v_n\}$$

where v_i is a variable and t_i is the term (variable or constant) with which we are going to replace v_i. It is possible to compose two substitutions by applying one to the other. □

Definition 2.1.4 A *query form* [BR86, HN84] is a conjunction of predicates written as

$$? - q_1(l_{11}x_{11}, \ldots, l_{1m_1}x_{1m_1}), \ldots, q_n(l_{n1}x_{n1}, \ldots, l_{nm_n}x_{nm_n})$$

where the label l_{ij} attached to the variable x_{ij} is empty, ? or ! and indicates that the variable is an existential, answer or an input variable, respectively. If x_{ij} is a constant then l_{ij} is empty. The input variables are bound with values supplied by the user, before the query is processed. When a query is processed, a set of ***answer tuples*** is built up and each tuple is a list of bindings for the answer variables. □

Definition 2.1.5 A Datalog *program* [CGT89] is made up of

- A set of Datalog rules.
- A set of query forms.
- A set of ground clause facts.

The set of ground clauses is called the *extensional database* (EDB) and predicates appearing in the EDB are called *extensional predicates*. The rule set is called the *intensional database* (IDB) and a predicate appearing as the head of a rule is an *intensional predicate* [Rei78]. We have the added restriction that a predicate cannot be both an EDB predicate and an IDB predicate. □

Note that in what follows I will often omit references to the set of ground facts (as is common in the literature, such as [CGT89]). The set is assumed to exist and is stored in a relational database. The above definition can therefore be seen as the definition of a *complete* Datalog program.

Example 2.1.1 A Datalog program that defines the predicate $path(From, To)$, which is the transitive closure of $link(From, To)$, is

$$path(From, To) : -link(From, To).$$
$$path(From, To) : -path(From, Z), link(Z, To).$$
$$? - path(?From, ?To).$$

The relation $link(From, To)$ is assumed to be stored in the database, while $path(From, To)$ is defined in terms of it, rather like a view in a relational database, except that *path* is recursive. □

Example 2.1.2 The following is an example of a Datalog program, along with one possible EDB. The intensional database is

$$\begin{aligned} ancestor(X,Y) &:- parent(X,Y). \\ ancestor(X,Y) &:- parent(X,Z), ancestor(Z,Y). \end{aligned}$$

and the extensional database is

$$\begin{aligned} &parent(tom, mary). \\ &parent(tom, john). \\ &parent(john, jane). \end{aligned}$$

The extensional database could be written in tuple form as:

$$parent = \{< tom, mary >, < tom, john >, < john, jane >\}$$

A non-ground fact such as $parent(fred, X)$ (Fred is the parent of everyone) cannot appear in the EDB. □

Several of the syntactic differences between Datalog and Prolog are due to the fact that the operational interpretation of a Datalog program is based on relational algebra (and relational databases). As such, a few syntactic restrictions arise from this, which I will now discuss.

The restriction that enforces the disjointness of the EDB and IDB predicates is not fatal as we can rewrite the program into an equivalent one which does have the disjointness property [BR86, CGM87]. To do this we *decompose* the program [BR86]. For every mixed predicate p we replace every fact $p(\overline{X})$ in the EDB with $p_E(\overline{X})$ and add the rule

$$p(\overline{X}) :- p_E(\overline{X}).$$

to provide the link between the two predicates.

A program is *safe* [BR86] [2] when it is guaranteed not to produce infinite answer relations. Even though the database may be finite, and the user-specified domain also finite, the use of evaluable predicates can lead to infinite relations being generated if variables are not restricted in some way. For example the rules

[2] There is some difference as to the actual definition. Ullman in [Ull88] says a program is safe when all the variables in the head are limited. However, this syntactic definition is equivalent to the semantic one given in [BR86].

$$NotEqual(X,Y) \ :- \ X \neq Y.$$
$$Likes(X,Y) \ :- \ Nice(Y).$$

both define infinite relations. A syntactic restriction which goes some way toward enforcing safety is range restriction which means that every variable in the head must appear in the body. This ensures safety if there are no evaluable predicates in the body of the rule. If we add that variables in evaluable predicates must also appear in base predicates then we get *strong safety* which does imply safety and is related to effective computability (see Section 2.3.1). This restriction is a bit too strong, so we allow variables in evaluable predicates to be connected by a chain of "=" to a variable in a base predicates. In other words the variable is limited.

Having no multiple occurrences of a variable or constants in the head of a rule will ease the evaluation of multiple rules in a predicate definition. To achieve this Ullman introduces the process of *rectification* [Ull88], which he confirms preserves safety and rule equivalence. In outline, rectification involves the following: For every constant "a" in the head we replace it in the head by a new and distinct variable "x_a" and add "$x_a = a$" to the body. For every repeated variable "x" in the head we replace it's ith occurrence by "x_i" and add "$x = x_i$" to the body. The DatalogIC system I have devised uses *full* rectification: constants and duplicates in the non-evaluable part of the body are also eliminated. Full rectification makes optimisation and conversion to SQL easier, as I will show in Section 3.3 and 5.3. Appendix A.3.1 gives the rectification algorithm I have used.

In summary, the differences between Datalog and Prolog are as follows:

- No functions are allowed as these can give rise to infinite relations.
- Negation is not allowed, although much research has already been done on this topic. See [She88] for a survey.
- Extra-logical features, such as the cut in Prolog, are not supported; in fact they are not needed since Datalog aims to be a pure declarative language.
- Clauses of the form $(p(\overline{X}) \ : -)$ that is, non-ground fact clauses in Prolog, are not allowed. All facts (which must have no variables in them) are stored in the database, and any predicate which appears in the database cannot appear as the head of a rule.

- The safety restriction is enforced in Datalog to ensure that there will be no infinite, and thus uncalculable, relations.
- More of a distinction is made between the ground facts, the EDB, and the rule set, the IDB.

The following two definitions are used later on.

Definition 2.1.6 An *integrity constraint* [CGM88] is a Horn clause of the form

$$q_1(\overline{X}_1), \ldots, q_m(\overline{X}_m) \rightarrow p(\overline{X})$$

where $p(\overline{X})$ can be an evaluable predicate. □

Integrity constraints constrain the set of allowable tuples in the database, and it is these that are used to represent semantic knowledge. A database is rejected if it fails to satisfy any of the constraints.

Definition 2.1.7 A conjunction C *subsumes* a conjunction D iff there is a substitution, δ such that $C\delta \subseteq D$. In other words if C is more "general" than D. If C subsumes D then we write $C \triangleright D$. □

2.2 Theory

In order to study Datalog programs, and to have a clear idea of what we are doing when we optimise a program, it is necessary to have a formal account of the meaning of a program. There are several ways of defining the semantics of Datalog programs, but the two most important are the declarative semantics and operational semantics as introduced in [Llo87]. The former is defined using the tools of mathematical logic (models and interpretations), while the latter is defined using fixed points and operators. It is important that these two interpretations be mutually consistent and that any implementation agrees with them. There are also two other views of Datalog programs: the model theoretic and proof theoretic models [Rei84]. We also need to define exactly what is meant by an answer to a query and, if we are to prove correctness results for optimisation algorithms, we need to define equivalence conditions.

2.2.1 Declarative Semantics

As with a Prolog program, a Datalog program can be viewed as an operator which takes as its input tuples for the extensional predicates and outputs tuples for the intensional predicates. The aim of this section is to describe this operator and to do so I will mention the key definitions and theorems (without proofs) from [Llo87, CGT89]. The main concept used is that of a Herbrand model for a Datalog program, and from this we get the least Herbrand model for the program which is taken to be the meaning of the program. We start off with the concept of an interpretation.

Definition 2.2.1 An *interpretation*, I, is a pair $< I_c, I_p >$ of mappings such that: I_c is a mapping of constant symbols to values for the constants, while I_p is a mapping of predicate symbols to relations such that the arity of p matches the arity of the relation.

A conjunction of predicates is *true* under an interpretation I if we can extend I_c to a mapping of terms (constants and variables) to constants (we will call this I_t) such that for every literal, $p_i(t_1, \ldots, t_n)$, in the conjunction, the tuple $(I_t(t_1), \ldots, I_t(t_n))$ is in the relation $I_p(p_i)$.

An interpretation I *satisfies* a rule of the form

$$p(\overline{X}) \quad :- \quad q_1(\overline{X}_1), \ldots, q_n(\overline{X}_n).$$

if whenever the conjunction $(q_1(\overline{X}_1), \ldots, q_n(\overline{X}_n))$ is true, then $p(\overline{X})$ is also true, under I.

We can extend this to satisfaction of a program: I satisfies a program if it satisfies every rule in the program. If so, then I is a *model* of the program. We write $S \models F$ if every interpretation that satisfies the set of rules S satisfies the rule F also. □

It has been shown that we can consider a special type of interpretation without any loss of generality.

Definition 2.2.2 The *Herbrand base*, H_P, for a program P is the set of all possible tuples for the predicates mentioned in P, i.e.

$$H_P \quad = \quad \{p(c_1, \ldots, c_n) \mid c_i \text{ is a constant and } p \text{ an n-ary predicate from } P\}$$

The constants are taken from the domains of the extensional predicates. We divide H_P into H_P^I for the intensional predicates and H_P^E for the extensional predicates. □

We can now derive from the Herbrand base a special type of interpretation.

Definition 2.2.3 A *Herbrand interpretation* for a program is an interpretation where I_p is a subset of the Herbrand base and I_c is simply the identity mapping. A literal, $p(t_1, \ldots, t_n)$, is true under a Herbrand interpretation, H, if we can map its variables onto constants such that the result is in H. Using Definition 2.2.1, we can also define when a Herbrand interpretation satisfies a program as well as define the Herbrand model.

An important type of Herbrand model is the *least Herbrand model* for a program P. This is the Herbrand model for P which is contained in every other Herbrand model of P. At the end of this subsection I give an existence theorem for the least Herbrand model of a program. □

Example 2.2.1 Consider the program

$$\begin{aligned} ancestor(X,Y) &:- parent(X,Y). \\ ancestor(X,Y) &:- ancestor(X,Z), parent(Z,Y). \end{aligned}$$

The Herbrand base for the above looks like:

$$\{ancestor(c_1,c_2), parent(c_1,c_2) \mid c_1 \text{ and } c_2 \text{ are constants from the domain of } parent\}$$

The following Herbrand interpretation is not a Herbrand model for the program

$$\{parent(a_1,a_2), ancestor(a_2,a_1)\}$$

since if we assume that the first rule is true under the interpretation, then using $parent(a_1,a_2)$ we conclude that $ancestor(a_1,a_2)$ is true under the interpretation, but it is not since it is not in the interpretation. The following two interpretations are Herbrand models:

$$\{parent(a_1,a_2), parent(a_2,a_3), ancestor(a_1,a_2), ancestor(a_2,a_3), ancestor(a_1,a_3)\}$$
$$\{parent(a_2,a_3), parent(a_3,a_3), , ancestor(a_2,a_3), ancestor(a_3,a_3)\}$$

Note also that the intersection of the above two models

$$\{parent(a_2,a_3), ancestor(a_2,a_3)\}$$

is also a Herbrand model. □

From here on I will be emphasising the distinction between the EDB and IDB. I do this because, unlike in Prolog programs, the EDB is often considered as the input to the Datalog program, which is the IDB part [Sag88].

Definition 2.2.4 For a program P and an EDB E, we define the set of *consequent* facts, $cons(P \cup E)$, as

$$\{F \in H_P \mid P \cup E \models F\}$$

which is the set of ground clauses which are satisfied by all Herbrand models of $P \cup E$. [CGT90] □

We are now in a position to state the key result of this section.

Theorem 2.2.1 *For every program P and EDB E*

$$cons(P \cup E) \;=\; \cap\{\mathcal{J} \mid \mathcal{J} \textit{ is a Herbrand model of } P \cup E\}.$$

Furthermore

$$cons(P \cup E) \;=\; \textit{Least Herbrand model of } P \cup E$$

The proof of this theorem can be found in [Llo87]. From the final statement of the theorem we can see that the declarative meaning of a Datalog program is its least Herbrand model.

2.2.2 Operational Semantics

In this section, I relate the least Herbrand model of a program to the operational meaning of the program. This is done in terms of operators and their least fixed points [3].

Definition 2.2.5 The *fixed point* of an operator T is an X such that $T(X) = X$. If we have a partial order, $\succ$, on the domain of T, then we can define the *least fixed point* as the X such that $T(X) = X$ and, for any other Y such that $T(Y) = Y$, we have $Y \succ X$. □

In our case our domain is $\mathcal{P}(H_P)$ which does have a partial order, namely $\subseteq$.

Definition 2.2.6 For every program P, we define the operator T_P which maps any subset of H_P, W, onto

$$W \cup \{p(\overline{X})\theta \mid p(\overline{X}) \text{ is the head of an } r \in P \text{ such that } \forall i \; q_i(\overline{X}_i)\theta \in W \;\}$$

□

[3] Note that I have omitted a lot of detail, such as continuity of operators and monotonicity, needed to prove Theorem 2.2.3

T_P can be viewed as one "iteration" of the program.

Theorem 2.2.2 *For any Datalog program P, the least fixed point of T_P with respect to an EDB E is the least Herbrand model of $P \cup E$, that is $cons(P \cup E)$.*

Theorem 2.2.3 *For any program P and EDB E, the least fixed point of T_P is equal to*

$$\bigcup_{i=0}^{\infty} T_P^i(E)$$

where T^i is T composed i times.

This is the key theorem which links together the operational semantics of a program and its declarative semantics. To complete the picture we define the operator onto which each program is mapped.

Definition 2.2.7 For a Datalog program P, we can define an operator $\mathcal{M}_P$ which is a mapping from the the power set of H_P^E to the power set of H_P. Using Theorem 2.2.3 we can define $\mathcal{M}_P(X)$ to be

$$\bigcup_{i=0}^{\infty} T_P^i(X)$$

Furthermore, if we are given a query Q along with the program, then we can define an operator $\mathcal{M}_{PQ}$ such that

$$\mathcal{M}_{PQ}(E) \;=\; \{H \mid H \in \mathcal{M}_P(E) \wedge Q \triangleright H\}$$

This gives us our first clue on how we should evaluate Datalog programs, a topic that is pursued further in Section 2.3. □

2.2.3 Other Semantic Views

As we aim to have a relational database system as a backend to our Datalog system we should be able to convert rules into relational algebra expressions. I will use the relational algebra as introduced in [Ull82]. I will just give a hint as to what this translation looks like, a more thorough presentation is given in [Ull88]. For the fully rectified rule

$$p(\overline{X}) \;:-\; q_1(\overline{X}_1), \ldots, q_n(\overline{X}_n), \delta$$

we get the relational algebra expression

$$R_p = \Pi_{\overline{X}} \sigma_{\delta'}((\ldots(R_{q_1} \bowtie_{\delta_2} R_{q_2}) \ldots \bowtie_{\delta_{n-1}} R_{q_{n-1}}) \bowtie_{\delta_n} R_{q_n})$$

where R_{q_i} is the relation corresponding to the predicate q_i, δ_i is that part of δ containing comparisons between variables from q_i and those from the result of $(\ldots \bowtie_{\delta_{q_{i-1}}} R_{q_{i-1}})$ and δ' is that part of δ containing comparisons with constants.

Example 2.2.2 Consider the rule

$$manager(X,Y) \quad :- \quad emp(X,D), dept(Y,D)$$

If the relation *emp* has columns *Ename* and *Dept* and the relation *dept* has columns *Mname* and *Dept*, then we can map the rule onto the relational algebra expression

$$manager = \Pi_{Ename,Mname}(emp \bowtie_{emp.Dept=dept.Dept} dept)$$

□

For recursive programs, the basic relational algebra is not powerful enough. We therefore have to augment the algebra with a fixed-point operation, and this is dealt with in Section 2.3. Section 5.3 shows how the DatalogIC system converts rules into an SQL expressions.

A final way of looking at a Datalog program is as a proof system composed of axioms and one or more inference rules. The importance of this view is that we can talk about the *proof* of an answer tuple, something that is done in Section 2.3.1. The Datalog program plus the extensional database, in ground clause form, and the standard axioms for the evaluable predicates used, make up the axiom set. The inference rule is simply *modus ponens*: we say a ground clause C can be inferred from a Datalog program P (written $P \vdash C$) if one of the following hold:

1. $C \in P$
2. There is a rule with head $p(\overline{X})$ and a grounding substitution θ such that $C = p(\overline{X})\theta$ and $P \vdash q_i(\overline{X}_i)\theta$ for every q_i in the body of the rule.

To determine the answer to a query, $Q(\overline{X})$, we check for which substitutions θ we have $P \vdash Q(\overline{X})\theta$. Each θ will be used to build one answer tuple .

2.2.4 Notions of Equivalence

If we are to develop algorithms which rewrite Datalog programs with the aim of making them quicker to evaluate, we have to be certain that the new program is equivalent to the old one. Sagiv in [Sag88] introduces some notions of equivalence, while in [CGM87] the concept of semantic equivalence is defined. A comprehensive study of equivalence is also given in [Mah88].

Definition 2.2.8 We say that two Datalog programs, P_1 and P_2 are *equivalent*,$P_1 \equiv P_2$, if for every extensional database E, we have $\mathcal{M}_{P_1}(E) = \mathcal{M}_{P_2}(E)$. □

It is possible to convert this into a statement about the models of the programs involved and from there devise algorithms to test for program equivalence [Sag88].

Definition 2.2.9 If we have a set of integrity constraints for a database, then we can speak about *semantic equivalence*. P_1 and P_2 are semantically equivalent, with respect to a set of integrity constraints I, $P_1 \equiv_I P_2$, if for every EDB, E, which satisfies I, we have $\mathcal{M}_{P_1}(E) = \mathcal{M}_{P_2}(E)$. □

Note that semantic equivalence is a weaker condition than straight equivalence and is implied by the latter. For a given program there will thus be more programs semantically equivalent to it than would be normally equivalent. This is the key reason for doing semantic optimisation—by restricting ourselves to databases which satisfy a set of constraints, we will have many more programs which are equivalent to the given one.

2.2.5 Studying Datalog Programs

As with all disciplines where objects are studied, it helps to classify the Datalog programs under study. With Datalog, the principle division is into non-recursive and recursive programs, while the primary tool of investigation is the graph. One benefit of dividing programs into classes is that it subdivides the problem of finding efficient evaluation and optimisation algorithms. Representing a Datalog program as a graph helps in determining which class of programs it belongs to, manipulating the program and evaluating it. The most common graphs used are the rule/goal graph and the adorned rule/goal graph [BR86]. The rule/goal graph shows the structure of the program in terms of which rules define which predicates (the goals), is used in determining which rules are recursive, and defines

an ordering for evaluation. The adorned graph shows the propagation of bindings through the program and is used in the Magic Sets algorithm (Section 2.4.2).

Definition 2.2.10 A *rule/goal graph* is a graph made up of two types of nodes: *rule nodes* and *predicate nodes*. There is an arc from a rule node to a predicate node if the rule defines the predicate, and there is an arc from a predicate node to a rule node if the predicate appears in the body of the rule. □

Using the rule/goal graph we can define a dependency relation between predicates [BR86].

Definition 2.2.11 A predicate p *depends directly* on another predicate q, written $q \rightarrow p$, if q appears in the body of a rule defining p. We can then look at the non-reflexive transitive closure of $\rightarrow$. A predicate p is *recursive* if there is a predicate q such that $p \stackrel{+}{\rightarrow} q$ and $q \stackrel{+}{\rightarrow} p$, while a rule is recursive if it defines such a p and has q in its body. In other words there is a cycle in the graph. If p is not the same as q then this is termed *mutual recursion*.

Using $\rightarrow$ we can draw up a *dependency graph* for the program. The nodes of this graph are the predicates of the program and there is an arc from p to q if $p \rightarrow q$. □

Example 2.2.3 For example, the rules:

$$r_1 \quad ancestor(X,Y) \ :- \ parent(X,Y).$$
$$r_2 \quad ancestor(X,Y) \ :- \ ancestor(X,Z), parent(Z,Y).$$

give rise to the rule/goal graph shown in Figure 2.1 (with abbreviated predicate names). The *ancestor* relation is defined by two rules: the first rule depends only on the *parent* relation, while the second depends on the *ancestor* and the *parent* relations. One can see that, as there is a cycle between *ancestor* and r_2, r_2 is a recursive rule and *ancestor* is a recursive predicate. □

Definition 2.2.12 A *reduced rule/goal graph* is a rule/goal graph which has had all nodes which are mutually recursive grouped into a single node. In graph-theoretic terms we identify the strongly connected components of the rule/goal graph and form the acyclic condensation of the graph.

In a similar way we can form the *reduced dependency graph* of the program. This will be used when we evaluate the program (Section 2.3). □

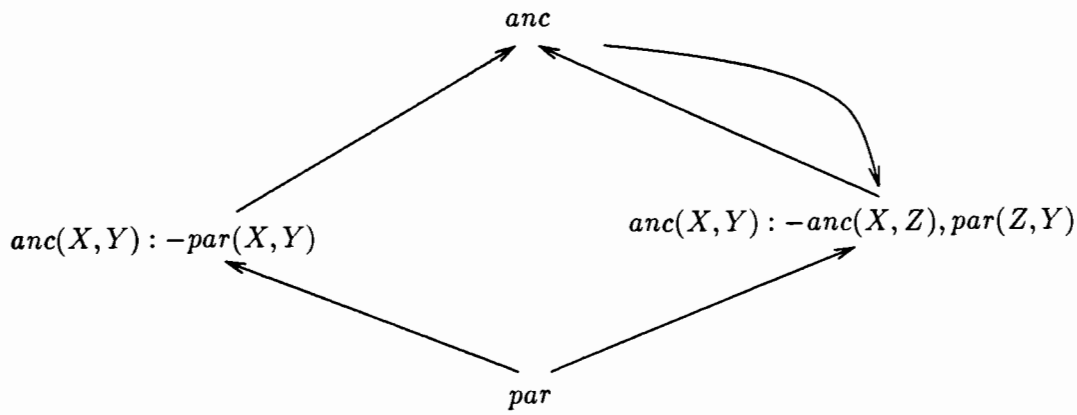


Figure 2.1: Rule/Goal Graph

For example, to form the reduced rule/goal graph for the program in Example 2.2.3 we would merge the node for r_2 with the node for the *ancestor* relation into one node.

Recursive programs can be divided into linear and non-linear recursive programs. A rule r, with head predicate p, is *linear* if there is only one q (which could be p) in the body of r such that $q \stackrel{+}{\rightarrow} p$ and $p \stackrel{+}{\rightarrow} q$.

The adorned rule/goal graph shows the propagation of variable bindings from a given query through the program. Such a graph contains *adorned* predicate nodes and rule nodes, several for each predicate and rule in the original graph [Ull85]. An adorned predicate is a predicate of the form p^a where a is a string of fs and bs. An f indicates that the variable in the corresponding argument position is free while a b indicates it is bound. So the possible adorned versions of *ancestor* are $ancestor^{ff}$, $ancestor^{bf}$, $ancestor^{fb}$ and $ancestor^{bb}$. To calculate the rule for an adorned predicate we take the original rule and propagate the bindings from the head through to the other IDB predicates in the body, assuming left-to-right evaluation. We do not adorn EDB predicates but they do help to propagate bindings, since if one variable in an EDB predicate is bound then they all are. When presented with a query such as $ancestor(X, tom)$ only part of the adorned graph is going to be reachable, so the rest can be ignored. The adorned version of Figure 2.1 reachable from the query $ancestor(X, tom)$ is shown in Figure 2.2

The algorithm to calculate the adorned rule/goal graph, and its corresponding program, is Algorithm A.6 in Appendix A.3.3.

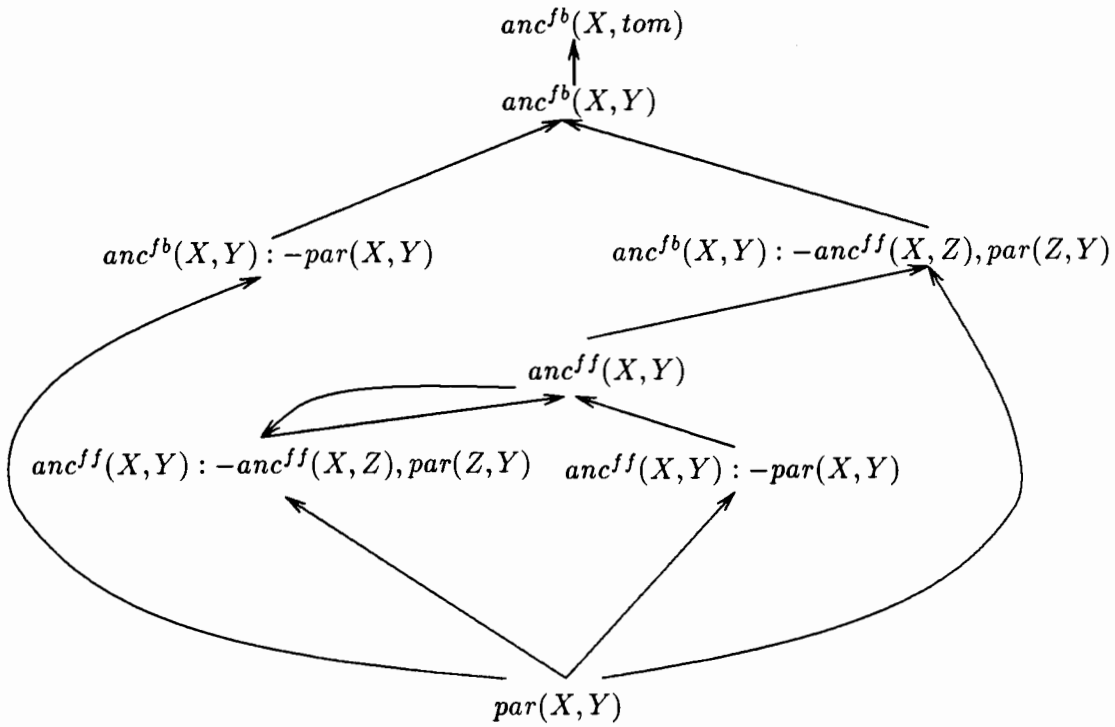


Figure 2.2: Adorned Rule/Goal Graph

2.3 Evaluation Schemes

In this section, I present a short overview of the concepts behind query evaluation. This is done since the desire for efficient evaluation of programs motivates the search for optimisation algorithms and, as noted below, evaluation and optimisation schemes are often two ways of looking at the same problem. I also present the best known scheme, Semi-Naive evaluation, which has been implemented in the DatalogIC system.

2.3.1 Ideas Behind Evaluation Schemes

In [BR86] it is pointed out that it is difficult to determine whether a technique is an optimisation or an evaluation scheme, and it is usually for pedagogical reasons that we call something an optimisation scheme when in fact it is often implemented as part of the evaluation phase. For example the Magic Sets method is presented as an optimisation scheme but can be built into the evaluation algorithm [HH87].

The precursor of all evaluation schemes is the least fixed point operator introduced in Section 2.2.2, and all systems should be compared against it when proving soundness and completeness. Evaluation schemes will differ from the fixed point operator with regard

to efficiency and the classes of programs to which they are applicable. It is generally the case that the more restrictive the class of programs that a scheme can handle, the more efficient it is. The Prolog top-down evaluation algorithm is also a measure against which we can compare systems.

The aim of an evaluation algorithm is to find answers to a query, which is a request for data from the database's store of knowledge. The *answer* to a query $Q(\overline{X})$, using a program P and EDB E, is the set of tuples

$$\{< a_1, \ldots a_n > \mid P \cup E \models Q(a_1, \ldots, a_n)\}$$

For each of these tuples there is a set of tuples which were used to *prove* it. We call these the *antecedent tuples*. The store of knowledge is composed of an extensional part, which comprises the facts explicitly represented, and an intentional part, which are the facts not held explicitly but which can be inferred from the rules. Simple classical databases lack the ability to store data using intentional definitions and so must set aside space for every fact (tuple) in its store of knowledge. Querying a classical database is thus primarily a fetch operation. Deductive databases, however, have a large virtual database defined by a set of rules. It would be unwise to generate this virtual database each time a query is presented and then handle the query as for a classical database. More sophisticated methods which take into account the query are therefore required. These methods will be goal-directed ones, where the query is the goal.

Types of Evaluation Scheme

In [BR86] a classification scheme involving three divisions for evaluation systems, is introduced.

Compiled vs interpretated: Compilation exploits the difference between the IDB and EDB. In many practical systems the extensional database will be a full DBMS, while the rule set will be much smaller and held separately. In the compilation phase only the rules in the IDB are accessed and these are compiled down to a `while` program [4], one for each query form. The `while` program will contain calls to the DBMS and loops to evaluate recursive components. The query form is a template marking out which arguments will be bound to constants at query time and which will be free and be part of the answer tuple.

[4] `while` programs are Pascal-like programs lacking procedures, functions and complex data structures.

At query time, when a query is presented, the corresponding program is run against the extensional database only. The advantage of this is that as much work as possible is done prior to when the queries are presented. With interpretation, the distinction between the two databases is lessened and both will be used at the same time, as in Prolog. The problem with compilation, as with all compiled languages, is that development time is slowed down by the compilation phase and if the program is simple, and not used often, more work may be done in compiling the program than executing it in an interpretive manner.

Top-down vs bottom-up: If we view the rules as being the productions of a grammar [5] we can use these to generate strings of EDB predicates which will map onto joins of EDB relations. With recursive rules we will end up with infinite strings. The aim of query evaluation is to find which terminal strings lead to the query string and then to evaluate these terminal strings. As in formal languages, we can do this in a top-down or a bottom-up manner. A top-down evaluation algorithm will work from the query down generating joins of terminal symbols. A bottom-up algorithm will start with the EDB predicates and generate relations upwards, using the query at the end to select the tuples required. Top-down is usually the more efficient, but more complex, while bottom-up is simpler, applicable to all programs, but less efficient since it does not use the query being presented until the very end. The Prolog algorithm is an example of a top-down algorithm while Semi-Naive (dealt with later) is a bottom-up system. However some optimisation techniques, presented later, go some way toward combining the "intelligence" of top-down with the simplicity of bottom-down (see Section 2.4.2).

Recursive vs iterative: This relates to whether the program (the compiled one or the interpreter) is recursive or iterative. With the latter the number of temporary relations is fixed while with the former they are not since the internal stack is used to hold temporary relations.

Effective Computability

Unlike the Prolog evaluation algorithm, the Datalog evaluation algorithms cannot handle infinite relations, and even if the program is safe (the answer relation finite) there may

[5] As pointed out in [BR86] the analogy is only a rough one as we have variables in the rules and the order of literals in the rules does not matter.

still be infinite intermediate relations. A program is *effectively computable* [BR86] if it can be guaranteed that there are going to be no infinite relations calculated during query evaluation. Having each head variable appear in a body predicate is often not enough to ensure effective computability as shown in Section 2.1.2. Strong safety will ensure a effective computability, as will a *bottom-up evaluable* [BR86] set of rules. A rule is bottom-up evaluable if it is range restricted and each variable in the body is secure, meaning that it either appears in a non-evaluable predicate or appears in an evaluable one in such a way that it will have an finite set of possible bindings. For instance [BR86]

$$p(X,Y) \quad :- \quad X > Y1, q(Y1,Y)$$

is not bottom-up evaluable since X is not secure.

Set of Potentially Relevant Facts

This concept, presented in [BR86], is useful in understanding how optimisation techniques work. The minimum set of facts needed to answer a query is the set of *relevant facts* and will be the set of facts used in the proof of the query. Viewed abstractly, the task of the query evaluation algorithm is find this set and trace a path through it to find the answers to the query. It would be ideal if the query optimisation technique could find this set first, but this is difficult to do without doing as much work as one would do in answering the query. Instead we work with the set of *potentially relevant facts* which contains the set of relevant facts. The better the evaluation algorithm, the smaller this set will be, which in turn will lead to fewer database accesses. The aim of an optimisation system is also to reduce the size of this set, but prior to query evaluation.

2.3.2 Naive and Semi-Naive Evaluation

Naive and Semi-Naive evaluation are perhaps the simplest of evaluation methods and, as these names suggest, they use little or no information about the structure of the query. They are bottom-up, compiled and iterative strategies although interpreted versions are known [BR86]. Here I will outline the framework of Naive and Semi-Naive evaluation, while Section 5.2 gives the algorithm for Semi-Naive as it is implemented in the DatalogIC system.

We cannot compile a recursive program down to a relational algebra expression involving EDB predicates only, as we will get infinite joins. It can be shown however that these

infinite joins can be evaluated finitely. So what we do is to build the joins up iteratively but at the same time evaluate them, exiting when there is no change in the current value for the relation. To do this we use a **while** program that has the following form:

current = Input EDB database.
while *current* changes **do**
    $current = current \cup T_P(current)$.
enddo

T_P is the operator introduced in Section 2.2 and can be viewed as being an operator that evaluates the join expression corresponding to the program P.

In reality we will have one such **while** loop for each recursive component in the program. During the compilation phase, the reduced dependency graph is used to generate a **while** program for each node containing one or more recursive predicates (the recursive components). These **while** programs are then concatenated together such that the program for a particular node can only be executed if the **while** programs for its antecedent nodes have been executed. This rule is called a *capture rule* a more detailed explanation of which is given in [Ull85].

The next stage is to build the operator T_P. The following system for evaluating recursive programs is due to [Ull88]. For every rule of the form

$$p(\overline{X}) \quad :- \quad q_1(\overline{X}_1), \ldots, q_n(\overline{X}_n).$$

we will have an evaluation statement of the form

$$R_p \quad = \quad eval(R_{q_1}, \ldots, R_{q_1})$$

where R_p is the relation associated with the head predicate and R_{q_i} the relation associated with the ith predicate of the body. This could be a relational algebra expression (see Section 2.2.3) or an SQL expression (see Section 5.3).

However, inefficiencies arise since if a tuple is proved in the ith iteration then, as its antecedent tuples are still in *current*, it will be proved again in each iteration after i; clearly this is a waste of resources. Semi-Naive evaluation aims to overcome these redundancies in the looping mechanisms by only calculating the increment between the old *current* and the new; thus the old tuples will not be recalculated. This relies on the fact that the

rules are monotonic [6], which depends on the absence of negation. We also have to find a formula for this increment which will take the form of a *difference equation*, which is used in place of *eval*( ). For a set of mutually recursive predicates, $\{p_i\}$ we will get a set of difference equations

$$R_{p_i} = eval\_incr(R_{p_1}, \Delta R_{p_1}, \ldots, R_{p_m}, \Delta R_{p_m}, R_{q_1}, \ldots, R_{q_n})$$

where p_i are the mutually recursive predicates, q_i are the predicates lower down the rule/goal graph (they are treated like EDB predicates but can be IDB predicates which have already been calculated) and ΔR_{p_i} is the difference between the old R_{p_i} and the new one. The `while` program will then loop though each R_{p_i} and evaluate its corresponding *eval_incr*() expression.

For linear programs we are fortunate in that

$$eval\_incr(R_{p_1}, \Delta R_{p_1}, \ldots, R_{p_m}, \Delta R_{p_m}, R_{q_1}, \ldots, R_{q_n}) = eval(\Delta R_{p_1}, \ldots, \Delta R_{p_m}, R_{q_1}, \ldots, R_{q_n})$$

For other types of program the difference equation will have to worked out for the program. We do know however that it must satisfy the following [BR86]

$$eval(\ldots, R_{p_i} \cup \Delta R_{p_i}, \ldots) - eval(\ldots, R_{p_i}, \ldots) \subseteq$$
$$eval\_incr(\ldots, R_{p_i}, \Delta R_{p_i}, \ldots) \subseteq eval(\ldots, R_{p_i} \cup \Delta R_{p_i}, \ldots)$$

Section 5.3 gives the complete Semi-Naive evaluation algorithm for linear programs that has been implemented in the DatalogIC system. Restricting ourselves to linear recursive programs is not a problem as most "real life" recursive situations are linearly recursive [BR86].

2.4 Optimisation

2.4.1 General Concepts

In this section, I will give some general ideas on optimisation which will serve as background to the discussion on optimisation in the rest of the dissertation. Included is the outline of one syntactic optimisation system, Magic Sets, which I have implemented.

Datalog is a declarative language and as such users are under no obligation to consider the efficiency of the rules they write. They should not be required to know the procedural

[6] In other words the current value of the IDB grows after each iteration of T_P.

meaning of the program. Optimisation should therefore be an integral part of any Datalog system. Optimisation can be described as

> The *efficient translation* of the rule set or query into one that is *equivalent* but *cheaper*, that is, requiring fewer resources to evaluate.

I now discuss each of the three words in italics. It is important that the optimisation scheme itself be *efficient*, otherwise if the time taken to optimise the query outweighs the original total query processing time, it is useless. Even if a scheme is exponential in the size of the rule set, the payoff may still be considerable since the rule set is often small and the database system, where the savings will be made, large. The value of the scheme will also depend on whether it is done once for the program when it is entered or each time a query is given. In order that the new program produce the same answers as the old one, the translation scheme must preserve *equivalence*, and in Section 2.2.4 I have presented two notions of equivalence that can be used. A query is *cheaper* when it consumes less resources. The cost for a query is composed of the following parts: intersite communication cost, secondary storage usage, main memory usage and processor time. Depending on the structure of the database system, these will carry different weights. For instance, in a distributed system intersite communication costs play a major role, while in a PC-based system, CPU and main memory usage play major roles. Various optimisation techniques deal with particular parts of the cost function and so are suited for systems where these parts are influential. It is generally felt that the join operation is the most costly in relational database systems, so any scheme which aims to reduce the number of literals in a rule is useful. However, there are systems which increase the complexity of the program and yet still produce a better program (see Section 2.4.2).

Jarke and Koch [JK84] have pointed out that there is a top-down and a bottom-up view of query optimisation. The latter came first historically and involves looking at specific types of query and optimising these without cognisance of any other types of program and general optimisation schemes. With the advent of the need for workable systems, this was found to be unsatisfactory and led to the top-down view which seeks to provide a general scheme, incorporating work done with the special cases.

Any optimisation scheme presupposes that we know the structure and semantics of a program, that we understand the interaction between rules (especially recursive ones), and that we understand the movement of data through the variables of a rule set. The

theoretic tools introduced in Section 2.2 are useful in this regard.

We can divide the set of optimisation schemes into the following classes:

- **Operational** These are low level schemes which rely on knowledge about the structure of the DBMS such as host machine type, file structure and distribution of data over networks—operational knowledge. I will not deal with this type of scheme, but [JK84] covers some of them.

- **Semantic** These schemes use knowledge at the other end of the spectrum—knowledge about the world being modeled. This knowledge is the semantic meaning behind the rules, and there can be more than one meaning attached to a set of rules. One way of representing this is with integrity constraints, of which functional dependences are a subclass.

- **Syntactic** These operate purely on the syntactic structure of the rule set and use no external knowledge. They are rewriting systems, either removing redundancies as in [Sag88], or adding extra rules and literals to enforce some behaviour as in the Magic Sets techniques [Ram88].

A more abstract view of an optimisation scheme is that it is a set of rules for manipulating the structure of a program, with or without a query, in order to reduce the resources required to evaluate the program whilst preserving equivalence. These rules and the two types of knowledge used by optimisation schemes, semantic and operational, can be put into a meta-database accessible to the scheme as suggested in [MZ87].

A general framework for an optimisation system, as given in [JK84], is as follows.

1. The query is translated into an internal representation, the type of which leaves the system as much freedom as possible to decide how the query should be evaluated and optimised. A declarative query language (such as relational calculus or Datalog) which, in its purest form, has no control structures, makes this task easier. Procedural languages will restrict the system since the user will already be imposing some evaluation scheme (which in some cases may be necessary).

2. The query is then transformed in order to reduce redundancies. This can be done, in the case of Datalog, in a purely syntactic manner as in [Sag88], or combined with semantic information as in [CGM87].

3. A set of evaluation schemes, or access plans, is created for the query making use of knowledge about the structure of the DBMS.

4. The cost for each evaluation scheme is calculated and the best one executed.

2.4.2 Magic Sets

One syntactic optimisation method I have implemented is Magic Sets [BMSU86, BR87, MFP90]. The magic set methods seek to reduce the set of potentially relevant facts by forming a "magic set" of elements. They do this by simulating the side-ways binding passing mechanism found in top-down algorithms such as Prolog. Some authors view them as rule augmenting optimisation techniques [BR86], while others see them as extensions to evaluation techniques where the operational semantics of the added rules is built into the evaluation technique [HH87]. From an implementation view the latter is better, but the former is easier to reason with.

We first rewrite the program into a reachable adorned program using the bindings from the query. We then build, using these adorned rules, the magic rules, which define the magic predicates and thus the magic set. Finally, we add the magic predicates to the front of the original rules.

Example 2.4.1 We illustrate the techniques with the canonical same generation example [BR86]:

$$\begin{aligned} sg(X,Y) &:- flat(X,Y). \\ sg(X,Y) &:- up(X,Z), sg(Z,W), down(W,Y). \\ query(X) &:- sg(a,X). \end{aligned}$$

First we generate the adorned program:

$$\begin{aligned} sg^{bf}(X,Y) &:- flat(X,Y). \\ sg^{bf}(X,Y) &:- up(X,Z), sg^{bf}(Z,W), down(W,Y). \\ query(X) &:- sg^{bf}(a,X). \end{aligned}$$

To generate the magic set we use the rules:

$$\begin{aligned} &magic(a). \\ magic(X) &:- magic(Y), up(Y,X). \end{aligned}$$

The first rule is obtained from the query, while the second is obtained from the recursive rule. From the rules we can see that the magic set will contain all the ancestors of a. We then add the magic predicate to the original rules:

$$\begin{aligned} sg(X,Y) &:- magic(X), flat(X,Y). \\ sg(X,Y) &:- magic(X), up(X,W), sg(W,Z), down(Z,Y). \\ query(X) &:- sg(a,X). \end{aligned}$$

We can see how the magic set reduces the set of relevant facts by first calculating the ancestors of a. This is what would have happened if the program was evaluated top-down. □

There is a related technique called Counting, which in the above example would record the length of each ancestor from a. Work has been done studying for which data and rule types these methods work best. Counting is the better of the two but more restrictive: it only works on data which is acyclic and programs containing linear rules, while Magic Sets works on bottom-up evaluable rules and any type of data. [SZ87] introduces Magic Counting methods which do overcome the above limitations. Section A.3.3 covers the implementation of Magic Sets optimisation in the DatalogIC system.

2.5 Conclusion

In this chapter, I have introduced Datalog, its syntax and semantics, and covered some of the theoretical underpinnings needed to study Datalog programs. I have also discussed the principals behind the evaluation and optimisation of programs. All of this will provide the foundations for the rest of the dissertation. Chapter 5 outlines the implementation of a basic Datalog system, covering algorithms for evaluation, simple syntactic optimisation and forming the various graphs for evaluating and optimising Datalog programs. The next chapter takes up where the section on optimisation left off, covering the main topic of this thesis, namely semantic optimisation.

Chapter 3

Semantic Optimisation Techniques

This chapter surveys the present state of semantic optimisation techniques. Semantic optimisation uses meta-knowledge about the data in the database to rewrite programs in order to avoid unnecessary computation where possible. In the first section of this chapter, I give some illustrative examples of what semantic optimisation does, the central ideas, some of the underlying theory and a brief survey of some systems. I will mention where ideas on semantic knowledge from "classical" database theory fit in with Horn clause form semantic knowledge, and also give a few of the problems associated with building the semantic knowledge base. In Section 3.2, I present the semantic optimisation scheme of Chakravarthy, Fishman, Grant and Minker [CGM87, CFM86, CGM90], which has as key concepts the ideas of residues and semantic compilation (I will label this the CGM system). In Section 3.3, I describe a slightly different system devised by Shenoy and Ozsoyoglu [SO87, SO89], which relies on the manipulation of graphs (I will label this the SO system). The following chapter will describe how semantic optimisation is carried out in the DatalogIC system, showing how the CGM and the SO systems can be linked together to form the single rule optimising component of the system.

3.1 Background

In this section, I will first illustrate what semantic optimisation entails with some motivating examples. I will then explain the ideas behind semantic optimisation, giving a survey

of some optimisation systems. The next two sections will show how semantic optimisation is carried out.

3.1.1 Motivating Examples

The following are some examples of what happens when semantic optimisation is applied to a query or rule. I hope that they also illustrate that semantic optimisation is an intuitively obvious thing to do. For each constraint, rule, or query I give the Horn clause form and its English equivalent. The examples use the following extensional relations:

- $emp(Name, Title, Department, Salary)$. The $Title$ column holds values like *manager*, *clerk* or *director*.
- $man(Manager, Department)$. This holds the names of managers and the departments they manage.

Example 3.1.1 Consider a database that satisfies the constraint

$$emp(Name, Title, admin, Sal) \rightarrow Sal > 5000$$

[All employees in *admin* earn more than 5K]

If we present the query

$$? - emp(?Name, Title, admin, Sal), Sal < 4000$$

[Find all employees in *admin* earning less than 4K]

to the database, then the system will return the answer that there are no tuples satisfying the query. Moreover, it can do this without having to search the database. □

Example 3.1.2 Assume that the *emp* relation is indexed on the column $Title$. If we have the query

$$? - emp(?Name, Title, Dept, Sal), Sal > 15000$$

[Find all employees earning more than 15K]

and the integrity constraint

$$emp(\_, Title, \_, Sal), Sal > 10000 \rightarrow Title = manager$$

[Any one earning more than 10K is a manager.]

then the optimiser will rewrite the query as

$$? - emp(?Name, manager, Dept, Sal), Sal > 15000$$

[Find all employees who are managers earning more than 15K]

Although the resulting query is slightly more complex, it will be an improvement since instead of scanning the *employee* relation from start to finish, the retrieval operation can make use of the index and consider only a portion of the relation. However, if the relation was not indexed on $Title$, then adding the restriction will not be profitable. This distinction is dealt with in Section 3.3. □

Example 3.1.3 This example is adapted from one in [JCV84] and illustrates how we can reduce the number of predicates, and thus the number of joins, in a query or rule. We have the following rule

$$same\_man(X, Y) : -emp(X, \_, D_1, \_), emp(Y, \_, D_2, \_), man(Z, D_1), man(Z, D_2)$$

[X has the same manager as Y if X is in department D_1, D_1 is managed by Z, Y is in department D_2 and D_2 is managed by Z.]

and the two integrity constraints

$$man(X, D_1), man(X, D_2) \rightarrow D_1 = D_2$$

[A manager manages only one department]

and

$$emp(\_, \_, D, \_) \rightarrow man(\_, D)^1$$

[1] The _ in the head of this constraint is short for "there exists a manager". See Section 3.1.3 for details.

[All employees have a manager]

Using these constraints we can optimise the rule to

$$same\_man(X,Y) : -emp(X,\_,D_1,\_), emp(Y,\_,D_1,\_)$$

[X has the same manager as Y if X is in department D_1 and Y is department D_1]

The first constraint equates D_1 and D_2, which leads to one occurrence of *man* being dropped. The second constraint then leads to the other occurrence of *man* also being dropped. □

Example 3.1.4 This example is different to the previous ones in that new semantic knowledge has to be inferred, something that is dealt with in Chapter 4. We have the constraints

$$emp(X, manager, \_, S_1), man(X, D), emp(\_, \_, D, S_2) \rightarrow S_1 > S_2$$

[The manager of a department earns more than the members of that department]

$$emp(X, \_, admin, S), \rightarrow S > 5000$$

[Members of the *admin* department earn more than 5K]

The query is

$$? - emp(?Name, manager, \_, S), man(?Name, admin), S < 4000$$

[Find all managers of employees in the *admin* department who earn less than 4K]

From the two constraints we can infer, using a transitivity rule, that

$$emp(X, manager, \_, S), man(X, admin) \rightarrow S > 5000$$

[All managers of *admin* employees earn more than 5K]

and thus optimise the query to FALSE, i.e. the query will have no answers. □

3.1.2 Concepts

As one can see from the above examples, semantic optimisation is by no means an unintuitive concept. Harking back to the days before computers it is something any intelligent archivist would be expected to do. When asked to retrieve a particular document, he would draw on past experience of what is in the archive and know where to look, or be able to reply politely that there is no such document, all without having to move from his desk. The archivist's past experience would have been determined by what the past requests have been, and this gives us a clue as to how the meta-knowledge base of an automated archiver could be built up. A related task is determining what counts as semantic information and how it should be structured. Logically, the only difference is that constraints can be more complex. For instance, they can have disjunctions, functions or variables in the head that do not appear in the body [Sag88]. Otherwise, both rules and constraints can be written in Horn clause form. The other difference is in their use: rules are used to generate facts, while constraints are used to test that the facts satisfy certain conditions. As an initial guide-line those parts of the database which are to be accessed the most should feature heavily in the semantic knowledge base. This implies that the database designer, as someone who is familiar with the global layout of the data, should build the meta-knowledge base. The query retrieval system can also keep a record of the traffic along the access paths through the database and perhaps automatically add constraints to the semantic knowledge base, the aim being to reduce the size of the search space for the most common queries. For this one would need a learning system that could derive general rules from examples. In Section 4.3, I give some tentative ideas on how the semantic knowledge can be structured and managed.

In Section 2.2.3 the proof theoretic view of databases was introduced. This can be extended to cover integrity constraints and semantic optimisation. Recall that from the proof theoretic view a Datalog program is a set of Horn clauses, which will be the rules of the program, the ground facts, and the axioms for the evaluable predicates (including equality). Query answering is a theorem proving activity where we attempt to prove the query from the database and obtain a set of bindings for the answer variables (the free variables). If we add integrity constraints then these will also be Horn clauses no different from rules. Thus in the evaluation of queries, which is done by trying to prove the query from the rules, the constraints will play the same role as the rules.

In the model theoretic view, a Datalog program is still a set of Horn clauses but the database is considered to be a model of the program, of which only a portion is actually stored (the extensional part). Query processing is testing the truth of a statement under this model, and during this process the initial model is extended and used to test the truth of the query. Integrity constraints have the role of constraining the set of allowable models for the program. They constrain the setting up of the initial model and thus the generation of the partial model during query answering.

3.1.3 Types of Meta-knowledge

In classical databases various types of semantic knowledge have been suggested, but these have been used mainly for data integrity checking purposes rather than optimisation [2]. They include functional dependencies, value bounds and subset constraints. Most of these older forms can be subsumed by Horn clause form integrity constraints, which can also give us generalised forms of the above three types.

Functional dependencies [Ull82] are used in relational databases to model the linkage between the attributes of a relation. A functional dependency for the relation R takes the form of $A_1, \ldots, A_k \rightarrow B_1, \ldots B_l$, where A_i and B_i are attributes in the relation R, and means that it is not possible to have two tuples that agree on values for attributes $\{A_i\}$ and disagree on values for the attributes $\{B_i\}$. As with other semantic knowledge, it is up to the database designer to decide which functional dependencies to impose. We can write functional dependencies in Horn clause form and thus incorporate them into DatalogIC programs. For the above functional dependency, we first rewrite it as a set of functional dependencies $\{A_1, \ldots, A_k \rightarrow B_i\}$, and then we convert each into the Horn clause

$$R(X_1, \ldots, X_n), R(X'_1, \ldots, X'_n), X_{a_1} = X'_{a_1}, \ldots, X_{a_k} = X'_{a_k} \quad \rightarrow X_{b_i} = X'_{b_i}$$

where the indices a_j are the column positions of each A_j and the index b_i is the column position of the attribute B_i. We require these complex indices since in relational algebra we work with attribute *names* while in logic programming we work with attribute *positions*. This difference will occur again when we convert rules into SQL statements.

Value bounds are easier to write in Horn clause form. Suppose we have a value bound, on relation R, of the form A *op* C where A is the name of an attribute from R and C is

[2] A notable exception is the use of functional dependencies to optimise queries via the tableau methods.

a constant. We can write this value bound as the clause $R(\ldots, X_a, \ldots) \rightarrow X_a \; op \; C$. This is a *local* value bound as it involves only one relation. We can also write non-local value bounds easily in Horn clause form. For instance

$$R(\ldots, X_a, \ldots), R'(\ldots, X_b, \ldots) \rightarrow X_a \; op \; X_b$$

is non-local value bound (provided $R \neq R'$).

Subset constraints, sometimes known as referential integrity constraints [JCV84], pose a problem since they can not be written directly in function-free Horn clause form. For the subset constraint $R.A \subseteq S.B$, which means that any value for column A in relation R will also appear in column B of relation S, we cannot write

$$R(\ldots, X_a, \ldots), S(\ldots, X_b, \ldots) \quad \rightarrow \quad dom\_subset(X_a, X_b)$$

since the meaning we would want to attach to $dom\_subset(X, Y)$ (the domain of X is a subset of the domain of Y) is not first-order.

$$R(X_1, \ldots, X_a, \ldots, X_n), X_a = Y_b \quad \rightarrow \quad S(Y_1, \ldots, Y_b, \ldots, Y_m)$$

will not work either since it is too strong: it says that if there is a tuple $< c_1, \ldots, a, \ldots, c_n >$ in R, then there is a *set* of tuples $\{< y_1, \ldots, a, \ldots, y_m >\}$ in S. In actual fact we want the Y_is (except Y_b) to be existentially quantified. The solution to this is to allow the constraints to have functions. This would allow us to use Skolem functions to model the existential quantifiers. The solution is thus

$$R(X_1, \ldots, X_a, \ldots, X_n), X_a = Y_b \quad \rightarrow \quad S(f_1(\overline{X}), \ldots, Y_b, \ldots, f_m(\overline{X}))$$

as pointed out in [CGM90]. Unfortunately not much is known about using constraints with functions, thus in DatalogIC we will do the following: We do not allow functions, but any "don't care" variable (ie "_") appearing in the head of a constraint is assumed to be existentially quantified. We can thus convert subset constraints into a form that can be used by the DatalogIC system.

Horn clauses provide a more general representation of semantic knowledge than is found in classical database theory. For instance, we can have the generalised subset constraint which has a condition which must be satisfied and involves more than one column. For example,

$$R(X_1, X_2, X_3, X_4), X_1 > 30 \quad \rightarrow \quad S(f_1(X_1), X_2, X_3)$$

which means that those tuples from R satisfying $X_1 > 30$ will have their second and third attributes appearing as the second and third attributes of some tuple in S.

3.1.4 Survey of Semantic Optimisation Systems

In this section, I give a survey of several semantic optimisation systems that have been proposed in the literature. None of these have been used in the DatalogIC system but did have a bearing on its development. Semantic optimisation has been approached both from an Artificial Intelligence perspective, where is it seen mainly as a knowledge management and utilisation problem, and from database theory where it is seen mainly as a query processing and an optimisation problem. Everyone agrees that both perspectives are vital. There is a third approach presented by Chakravarthy, Grant and Minker. This is a logic-based approach, which I present in Section 3.2.

The first system I look at is the QUIST (QUery Improvement through Semantic Transformation) system developed by King [Kin81]. He takes the knowledge management approach and points out that there are in fact two types of knowledge, which will be interacting. We first have knowledge of the semantics of the application, and we also have knowledge of the structure and processes making up the database system itself. We use the latter knowledge to control the application of the former in rewriting a query. This has to be done if we wish to control the number of transformations made and avoid a costly optimisation system.

QUIST was designed for relational databases and the class of queries that it can handle is a subset of the restrict-join-project type. The system works by means of the generate-test approach, which involves a search-space (each point being a query equivalent to the original) and a set of transformation rules for moving from one point to another. There are three levels of operation, each guided by their own set of heuristics. The first level is the planning stage which considers the query abstractly trying to obtain a list of constraint targets (relations mentioned in the query). During the second stage (the generation stage) QUIST moves through the space of semantically equivalent queries. Each transformation is based upon the inference of new constraints which are added to the query. Finally we have the testing stage where QUIST moves this time though the space of physical realisations of the query in order to find the cheapest one to execute. During this stage, the query is being considered the least abstractly.

A set of heuristics introduced by King bear further study and some have been built into the SO system given in Section 3.3. They are basically rules on the type of transformation that will give us a better query, and are based on operational considerations:

1. Index Introduction—Try to introduce a restriction on an indexed attribute.

2. Join Elimination—Joins are expensive so try to eliminate them.

3. Scan Reduction—Apply a restriction on a relation involved in a join to reduce the number of scans, of the second relation, for matching tuples.

4. Join Introduction—Introduce a join on a small relation . This can help to speed up the calculation of joins on larger ones.

A heuristic is also introduced which tests to see if it is worth optimising a query in the first place.

In [JCV84] Clifford, Jarke and Vassiliou describe a system which is written in Prolog and acts as an optimising front-end to a relational query system. Their idea is to have a Prolog expert system front-end loosely coupled to a database which implements the SQL language. The translation between the two is done via a meta-language DBCL (Database Call Language), which is also the medium used for optimisation. A local optimiser performs syntactic and semantic simplification while a global optimiser stores multiple database calls, thus overcoming the tuple at a time/set orientated difference between Prolog and SQL. The meta-language is used to represent the database schema, query and semantic knowledge. The latter includes value bounds, functional dependencies and referential integrity constraints. Prolog queries and views are translated into a tableau-like representation which, after optimisation, is used to generate the SQL query. The optimisation phase includes syntactic as well as semantic simplification. Semantic transformations are done first, and they involve the adding of implied value bounds, using functional dependencies to reduce the number of rows in the tableau (by means of well-known algorithms as in [Ull82]), and using referential integrity constraints to eliminate dangling rows. They also propose a system for the inference of referential constraints. Possible extensions that they suggest include the handling of disjunctions, negation and recursion. The approach taken is a database one utilising well known database techniques and algorithms. The advantage of this approach is that efficiency and soundness of each separate module is en-

sured; however, interaction between them is restricted and generality is difficult to achieve. The next system proposed overcomes this, as does the logic approach of [CGM87].

A similar system to the previous one is presented by Jarke in [Jar86]. The major difference is that a graph-theoretic representation is used instead of a tableau one. As in the above system, a Prolog front-end is loosely coupled to a database system and interaction done via an intermediate language. Jarke notes that there are two types of semantic knowledge. The first type are specific constraints about small sets of data. The size of this collection of knowledge is proportional to the size of the database, so it can become large and AI-type heuristics are required. This is the type of knowledge that the QUIST system uses. The second type is knowledge found in database systems: general laws applicable to all elements of a relation or combination of relations. This type of knowledge is easy to recognise, apply and infer from. This distinction is important since it influences the approach taken and thus the design and efficiency of the system.

In view of efficiency constraints, the Jarke system will only accept the usual database-type semantic knowledge, as in the previous system. The architecture is similar to the above except that a knowledge base and blackboard representation is used. The tableau used in the above system is replaced with a graph representation for the functional dependencies and one for the value bounds. The main difference between this system and the last one is that an attempt is made at merging the algorithms and data structures. This will enable there to be interaction between the optimiser that uses functional dependencies and the one that uses value bounds. We can also achieve this using the logic-approach of [CGM87] as both functional dependencies and value bounds can be written in Horn clause form. Semantic optimisation can also provide meaningful error messages to the user who enters queries with empty answers. The optimiser will find the constraints that cause the query to be FALSE and can tell the user why the query failed.

Other semantic optimisation systems have been given in [HM75, ASU79, HL88]. The last of these deals with optimisation of recursive programs, something I deal with in Section 4.4.

In summary, one can derive several principles and concepts from the reports on previous semantic optimisation systems. There are three approaches that can be taken: the artificial intelligence approach which uses knowledge base techniques and heuristics, the database approach which uses well-known algorithms for dealing with familiar types of semantic

knowledge, and finally the logic-based approach (described in Section 3.2) where a logic representation along with resolution and unification algorithms is used. As suggested in [CGM90] the logic approach should aim to formalise the AI approach and turn the heuristics into logical laws, and it should subsume the database approach showing that there is a logical equivalent to each type of semantic knowledge and each algorithm used. Another interesting point is whether each different type of semantic knowledge should be treated separately and be used to optimise the query individually or should be merged so that interaction is possible. To achieve the latter, a common representation is an advantage, again making a logic-based approach seem worthy of investigation.

With any optimisation system the final payoff depends on what is actually passed to the optimiser and what the structure of the data is. It is possible to calculate an approximate gain. For instance knowing how much a join costs, we can calculate the benefit gained for any optimisation that deletes joins. The real cost saving of each optimisation can only be calculated once the true cost of each operation on the particular database is calculated and most of the time this can only be done once the operation has been carried out. In [BR86] some analytic work as been done on comparing evaluation and optimisation techniques on artificially structured databases, such as trees and cylinders, and in [SO89] some analytic, as well as empirical, indication is given of the gains of their system. Section 5.4 covers some of the results I obtained for the DatalogIC system.

3.2 Semantic Query Optimisation

3.2.1 Introduction

The first semantic optimisation scheme I look at is the semantic query optimisation system of Chakravarthy, Fishman, Grant and Minker [CGM87, CFM86, CGM90]. This will form the first part of the rule optimisation module I develop later. Only those concepts that are used in the DatalogIC system are dealt with in any detail and the notation has been changed to conform with that used throughout the rest of this dissertation.

The architecture that they propose is illustrated in Figure 3.1 [CGM87]. The semantic compilation module (SCM) takes the set of rules (the IDB) and, using the set of integrity constraints, generates a set of *semantically constrained axioms*. The constrained axioms will be the original rules plus a set of *residues* for each. A residue of a rule and constraint

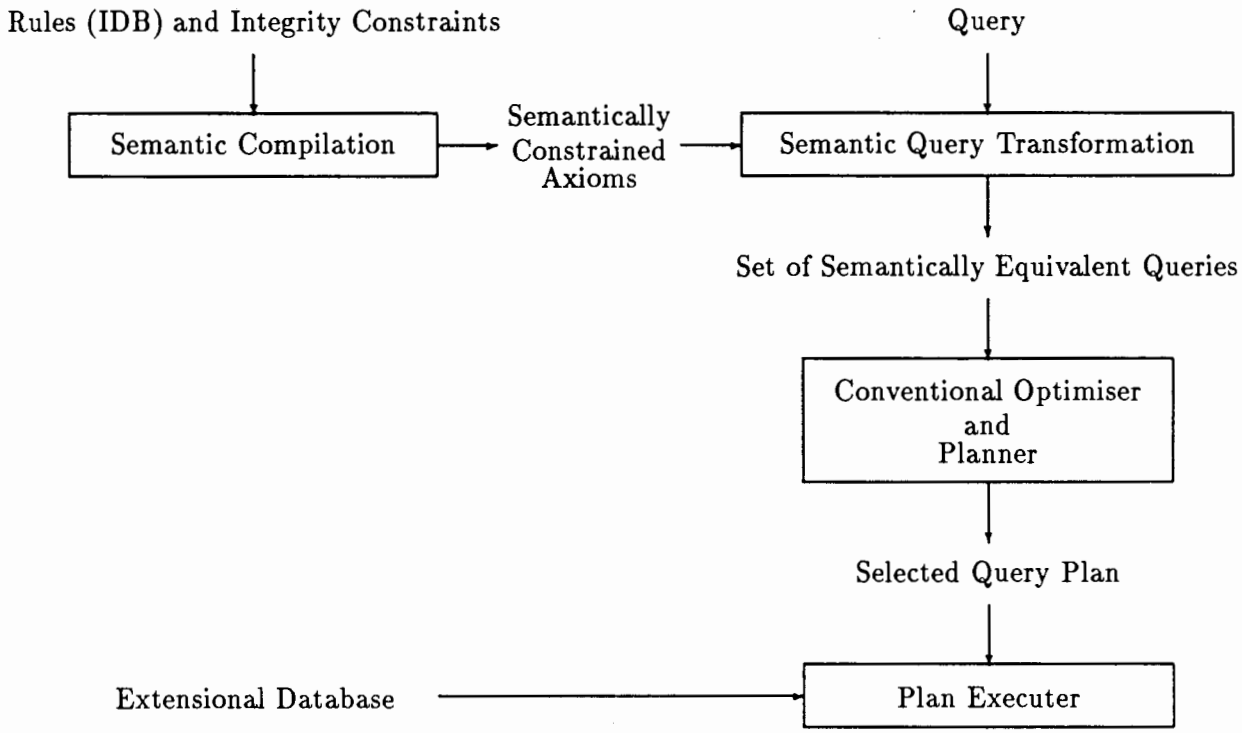


Figure 3.1: Query Processing Using Semantic Query Optimization

is the subclause of the integrity constraint that does not subsume the body of the rule. The semantic query transformation module will take a query and, using the rules, compile it down to an expression involving base relations (thus only non-recursive programs are permitted). During this compilation process any residues attached to rules used will be added to the residue set for the query. These residues are used to optimise the query into one or more semantically equivalent, but hopefully, cheaper queries. The conventional query optimiser will then apply operational knowledge and heuristics to optimise each query further, passing it to the plan executer which will pick the cheapest plan and execute it against the extensional database. The following example illustrates what occurs at each stage using Example 3.1.3 from the previous section.

Example 3.2.1 Recall that the constraints are as follows

$$emp(\_,\_,D,\_) \rightarrow man(\_,D)$$
$$man(X,D_1), man(X,D_2) \rightarrow D_1 = D_2$$

and the rule is

$$same\_man(X,Y) \ :- \ emp(X,\_,D_1,\_), emp(Y,\_,D_2,\_), man(Z,D_1), man(Z,D_2), X \neq Y.$$

The SCM will output the rule plus its set of residues which are as follows:

$$\rightarrow man(\_, D_1). \quad \rightarrow man(\_, D_2). \quad \rightarrow D_1 = D_2.$$

Note that the variables in the residues are no longer the ones that the predicates had in the integrity constraint but are ones from the rule, and that the first integrity constraint gives us two residues. The query

$$? - same\_man(?Name, john)$$

is compiled down to

$$? - emp(?Name, \_, D_1, \_), emp(john, \_, D_2, \_), man(Z, D_1), man(Z, D_2)$$

and optimised to the following

$$? - emp(?Name, D_1), emp(john, D_1)$$

using the method to be described in Section 3.3. □

The DatalogIC system uses only the semantic compilation part of the CGM system, details of which are given in the rest of this section. Section 3.3 will show how the residues are used to optimise queries and rules by the SO system.

3.2.2 Calculation of Residues

The most interesting and useful concept introduced in [CGM87] is the idea of residues. This section will give the formal definition of residues along with the algorithm used to calculate them. In the DatalogIC system, residues are passed to the SO system to be used to optimise the rules of a program.

The semantic compilation module takes as input a set of rules and a set of integrity constraints. As well as having the usual clause form, the constraints and rules must satisfy the following conditions (some of which have been mentioned in Chapter 2) [CGM87]:

1. A predicate cannot appear in the EDB and be the head of a rule.

2. Rules are range-restricted and there are no constants or repeated variables in the head (i.e. they must be rectified).

3. Integrity constraints must contain only occurrences of EDB predicates in the body or contain IDB predicates that can be replaced by a finite string of EDB predicates. In other words, we must be able to compile a constraint I into one mentioning only EDB predicates. We call the compiled constraint I^*.

4. Constraints must be range-restricted.

A database which satisfies conditions 1 and 3 is called *structured* and [CGM87] gives an algorithm to convert any database to a structured one. The DatalogIC system enforces 1 and 4, and ensures that 2 is true (by rectification).

In order to calculate residues we use the notion of subsumption defined in Definition 2.1.7. To determine if a clause C subsumes another clause D we use a resolution-based algorithm: we first ground D, by replacing variables with new and unique constants (using a substitution θ), negate it, and then carry out linear resolution with C as the root and the literals of $\neg(D\theta)$ as the input clauses. If a contradiction results, then C subsumes D. However, it may be that only a part of C subsumes D; in this case we would like to know the subclause of C that does not subsume D—this will be the residue. To calculate residues we have to put the integrity constraint into *expanded* form which is very similar to full rectification outlined in Section 2.1.2 [3]. The reverse of expanded form is *reduced* form, but this is not used in the DatalogIC system. For a clause C, we call its expansion C^+ and its reduction C^-. The following example from [CGM87] shows why full rectification of the constraint is necessary.

Example 3.2.2 We have the constraint

$$r(X, a) \rightarrow$$

and the rule

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

Intuitively, we can see from the constraint that the rule is redundant [4] if Z is ever bound to a, for instance, when we pose the query

$$? - p(X, a)$$

[3] The difference is that in [CGM87] evaluable predicates are treated differently.

[4] i.e. the rule will never produce any tuples—see Section 4.5.1 for more details on what happens when a rule is made redundant

However, when we apply the subsumption algorithm it will ground the rule to

$$p(a_x, a_z) : -r(a_y, a_z), s(a_x, a_y)$$

This prevents $r(X, a)$ from unifying with $r(a_y, a_z)$ in the rule and so the algorithm will return FALSE which is not the result we want. The solution is to take the constant a out of the first r, which can be done by full rectification or expansion of the constraint which will look like

$$r(X, X_a), X_a = a \rightarrow$$

The subsumption algorithm will then tell us that the residue is

$$Z = a \rightarrow$$

which means that whenever Z is bound to a the rule is redundant. □

Definition 3.2.1 A subclause D of a clause C is *nep-maximal* with respect to a property P if D has property P, and for every clause D' such that $D \subseteq D' \subseteq C$, which has property P, D contains all the non-evaluable predicates of D'. □

Definition 3.2.2 A *residue* of an integrity constraint, I, and a rule A is:

$$(((I^+ - I_m)\delta))\theta^-$$

where I_m is a nep-maximal subclause of I^+ which subsumes the body of A, θ is a substitution which reduces A to a ground clause, θ^- is the inverse of θ, and δ is the substitution which makes I_m an instance of A. □

If the whole of I^+ subsumes A (i.e. $I_m = I^+$) we get the *null residue*, while if no part of I^+ subsumes A (i.e. $I_m = \emptyset$) we get the *maximal residue* I^+ itself. Between these two extremes we will obtain a set of residues some of which may be *redundant* residues if the head of the residue is TRUE or if the body is FALSE.

Definition 3.2.3 An integrity constraint is *merge compatible* with an axiom if there is at least one residue that is non-redundant and non-maximal. □

The merge-compatible constraints are the important ones as they produce residues which can be used to optimise a query.

Example 3.2.3 This example illustrates the various structures defined above. Consider the rule

$$p(X,Y,Z) \ :- \ q(X,Z), r(Z,Y), X > 3, Y = 4.$$

and constraint I

$$q(A,B), r(B,C), B > 5 \ \rightarrow$$

The expanded form I^+ of the constraint is

$$q(A_0,B_0), r(B_1,C_1), B_0 = B_1, B_0 > 5 \rightarrow$$

The nep-maximal subclause of I^+ that subsumes the body of the rule, I_m, is

$$q(A_0,B_0), r(B_1,C_1) \rightarrow$$

which means that the residue of I and the rule is $(Z > 5 \rightarrow)$ [5]. On the other hand $(r(Z,Y), Z > 5 \rightarrow)$ is not a residue since $r(Z,Y)$ is a part of I_m. If we had the constraint

$$s(A,B) \rightarrow$$

then the residue of this and the rule will be the maximal residue

$$s(A,B) \rightarrow$$

Finally, consider the constraint

$$q(A,B), r(C,D) \rightarrow C = B$$

The residue of this and the rule is $(\rightarrow B = B)$ which is redundant since $B = B$ is TRUE. If we did not have $C = B$ in the head of the constraint then the residue would be $(\rightarrow)$, which is the null residue. □

The algorithm to calculate residues, given in Figure 3.2 and used in the DatalogIC system, follows closely that found in [CGM87]. Note: θ is the substitution that grounds the rule A and *nc_in_rec* is the flag that is set by the *Resolve* function if the null clause comes out as the result of a resolution.

[5] $Z = Z$ is also in the residue but is deleted as being redundant.

Algorithm: Calculate residue set of a rule and integrity constraint
INPUT: Rule, $A = \{p : -q_1, \ldots, q_n\}$, and integrity constraint, I
OUTPUT: If I is merge compatible with A, then the set of residues is returned;
Else if I fully subsumes A then **FULLY_SUBSUMES** is returned.
Otherwise **NOT_COMPATIBLE** is returned.

```
BEGIN
    unit_literals     = {q1θ, ..., qnθ}.
    fully_subsumes    = FALSE.
    not_merge_comp    = TRUE.
    nc_in_res         = FALSE.
    new_res           = {I+}.
    residues          = ∅
    while new_res <> ∅ and not nc_in_res do
        resolvants = new_res.
*       new_res = ∅.
        For each unit literal, L, in unit_literals
            For each clause, C, in resolvants
                If L was not used in previous resolutions to obtain C then
                    new_res = new_res ∪ Resolve(L, C).
        For each clause, C, in resolvants
            If C was unused then add it to residues.
    end while

    If nc_in_res is true then
        fully_subsumes = TRUE.
    else if residues = {I+} then
        not_merge_comp = TRUE.
    else for every clause, C, in residues
        Apply θ^-1 to C and simplify it.
        If C is false then
            fully_subsumes = TRUE.
        If C is not redundant then
            not_merge_comp = FALSE.
        else
            Delete C from residues.
        endif
    endif
    Return correct code depending on values of fully_subsumes
        and not_merge_comp.
END
```

Figure 3.2: Residue Calculation

Example 3.2.4 This examples shows how the residues in Example 3.2.1 are calculated. Using the first integrity constraint we get:

At statement * for the first time:

$$\begin{aligned} unit\_literals &= \{emp(a_x, \_, a_{d1}, \_), emp(a_y, \_, a_{d2}, \_), man(a_z, a_{d1}), man(a_z, a_{d2})\} \\ resolvants &= \{\ emp(\_, \_, Dept_1, \_), Dept_1 = Dept_2 \rightarrow man(\_, Dept_2)\ \}. \end{aligned}$$

At * after iteration 1:

$$\begin{aligned} resolvants &= \{\ (a_{d1} = Dept_2 \rightarrow man(\_, Dept_2)),\ (a_{d2} = Dept_2 \rightarrow man(\_, Dept_2))\ \} \\ residues &= \emptyset. \end{aligned}$$

At * after iteration 2:

$$\begin{aligned} resolvants &= \emptyset. \\ residues &= \{\ (a_{d1} = Dept_2 \rightarrow man(\_, Dept_2)),\ (a_{d2} = Dept_2 \rightarrow man(\_, Dept_2))\ \} \end{aligned}$$

Using the second integrity constraint we get:

At statement * for the first time:

$$\begin{aligned} unit\_literals &= \{\ emp(a_x, \_, a_{d1}, \_), emp(a_y, \_, a_{d2}, \_), man(a_z, a_{d1}), man(a_z, a_{d2})\ \} \\ resolvants &= \{\ man(Man_1, Dept_1), man(Man_2, Dept_2), Man_1 = Man_2 \\ &\qquad \rightarrow Dept_1 = Dept_2\ \}. \end{aligned}$$

At * after iteration 1:

$$\begin{aligned} resolvants = \{\ &(man(Man_1, Dept_1), Man_1 = a_z \rightarrow Dept_1 = a_{d2}), \\ &(man(Man_2, Dept_2), Man_2 = a_z \rightarrow Dept_2 = a_{d1})\ \} \\ residues &= \emptyset \end{aligned}$$

At * after iteration 2:

$$\begin{aligned} resolvant &= \{a_z = a_z \rightarrow a_{d1} = a_{d2}\}. \\ residues &= \emptyset. \end{aligned}$$

At * after iteration 3:

$$\begin{aligned} resolvants &= \emptyset. \\ residues &= \{a_z = a_z \rightarrow a_{d1} = a_{d2}\}. \end{aligned}$$

After the removal of redundancies, such as $a_z = a_z$, and replacement of the constants by variables we get the three residues

$$(\rightarrow D_1 = D_2), (D_1 = Dept_2 \rightarrow man(\_, Dept_2)), (D_2 = Dept_2 \rightarrow man(\_, Dept_2))$$

□

The following theorem, which is proved in [CGM87], shows that the above algorithm is sound and complete.

Theorem 3.2.1 *Given a rule A and constraint I, the above algorithm generates only the non-redundant residues and if I fully subsumes A or is not merge compatible with A, then the correct value is returned.*

The next theorem confirms that using residues is a logically sound procedure. The theorem is also used, along with the theorems of correctness for the SO system, to prove that the system I propose does produce a query which is semantically equivalent to the original one.

Theorem 3.2.2 *Every residue of I and A is a logical consequence of* $I \cup \{q_1, \ldots, q_n\}$, *where* q_i *is a literal in the body of A.*

3.2.3 Semantic Query Transformation

The semantic query transformation phase is when residues are used to rewrite a given query Q. Depending on the form that a residue takes, it can be used in one of four ways to optimize Q:

1. If it is NC, the null clause, then we can reject Q immediately.

2. If it is a goal clause $(A \rightarrow)$ and A is evaluable, then we can negate A and add it as a new restriction to Q. If A is not evaluable and A subsumes a literal in Q, then again the query is FALSE.

3. If it is a unit clause $(\rightarrow A)$ and if it is evaluable, we add it as a new restriction to Q. However if it is not evaluable, then it may be advantageous to add it to the body of Q even though there will be an extra join.

4. If it is an implicational clause ($A \rightarrow B$), then it may be possible to add B to Q if A subsumes a literal in Q.

In the DatalogIC system, case 4 and the second part of cases 2 and 3 are not used. However, the system I propose in Chapter 4 is no less powerful because of this, as I will show in Section 4.5.2. The recent paper [CGM90] gives more ideas on the semantic query transformation phase. In the next section I will outline the SO system which uses the residues generated by the algorithm above and which can be seen as an implementation of the semantic query transformation phase presented in [CGM87].

3.3 Using Residues

This section contains ideas from [SO87, SO89] but uses a more formal presentation and makes explicit underlying definitions and theorems from these papers. As presented in [SO89] the system is designed for relational databases and takes a conjunctive query and constraints in *predicate.attribute* form.

Example 3.3.1 The SO system accepts queries like

$$? - manager.Dept = admin, manager.Name = emp.Name, emp.Sal > 15$$

and constraints like

$$manager.Name = smith \rightarrow manager.Dept = admin$$

□

However, since the constraints in the DatalogIC system are in Horn clause form and the system is going to be used to optimise rules, I have changed the notation and terms used. Section 4.5.2 gives more details of this transformation.

The first part of the section describes the various transformations a rule undergoes without making any commitment on how these transformations are to be carried out. Only the properties that each transformation must satisfy are given. In the second part proofs of completeness and soundness are given, confirming that the transformations preserve equivalence. In Appendix A.4 I give *one* way that these transformations can be carried out, the one given in [SO89] and implemented in the DatalogIC system.

3.3.1 Mechanics of Residue Application

The SO system revolves around the use of graphs which represent the interconnection of the comparison predicates in a rule body. The first graph, the conjunction graph [6], is a direct graphical representation of the comparison operator part of the rule. The vertices are the variables and constants that appear in the rule and each edge represents a comparison operator from the rule. Since the rule is fully rectified the graph will also hold information about joins in the edges between attributes. This graph is converted to a condensed canonical form which has the property that any two equivalent rules will be reduced to the same canonical graph. We also remove redundant edges and check for consistency. Up to this point we have not used the integrity constraints at all. The next step is to form the semantic expansion of the graph. This is done by adding edges, implied by the graph and integrity constraints, to create what can be seen as a semantically maximal graph. We also have to ensure that the graph is still in condensed form. The next two stages aim at removing redundant relations (and thus nodes) and non-profitable, redundant edges. The final stage is the conversion back to rule form.

Example 3.3.2 This example is used throughout the section to illustrate what happens at each stage. We shall be using the following extensional relations

- $flight(PilotName, From, To, Distance)$
- $emp(Ename, Class, Salary)$
- $man(Mname, Ename)$

We have the integrity constraints:

- All flights from Moscow are more than 3000km long.
- Any pilot who flies a journey of more than 2000km earns more than 50K.
- Only pilots fly.
- Only employees are managed.

In Horn clause form these constraints look as follows:

[6] This is called the query graph in [SO89], but since we are working with the body of a rule I have generalised the name.

$$flight(\_, moscow, \_, Dist) \rightarrow Dist > 3000.$$
$$emp(Emp, pilot, Sal), flight(Emp, \_, \_, Dist), Dist > 2000 \rightarrow Sal > 50.$$
$$emp(Emp, Class, \_), flight(Emp, \_, \_, \_) \rightarrow Class = pilot.$$
$$man(\_, Emp) \rightarrow emp(Emp, \_, \_)$$

The query:

> "Find all managers of pilots who fly journeys of more than 1000km from Moscow."

is represented in Horn clause form as:

$$?- \quad man(?Man, Emp), emp(Emp, pilot, Sal), flight(Emp, moscow, \_, Dist), Dist > 1000.$$

Next we translate the above constraints into the form that can be accepted by the system. Section 4.5.2 describes how this is done.

$$flight.From = moscow, flight.Dist \leq 3000 \rightarrow$$
$$emp.Class = pilot, emp.Emp = flight.Emp, flight.Dist > 2000, emp.Sal \leq 50 \rightarrow$$
$$emp.Class \neq pilot, emp.Emp = flight.Emp \rightarrow$$
$$man.Emp \subseteq emp.Emp$$

These are then used to optimise the query. Note that in order to distinguish between variables in the rule that are the same, but appear in different predicates, I use the *predicate.variable* notation. □

From Rules To Conjunction Graphs

Given a rule that we wish to optimise, we convert this into a conjunction graph. The graph is useful as it brings out the various transitivity properties of the evaluable predicates. Since the rule is fully rectified, we only have to consider the comparison operator part of the rule when converting to graph form.

Definition 3.3.1 For the fully rectified rule

$$p(\overline{X}) \quad :- \quad q_1(\overline{X}_1), \ldots, q_n(\overline{X}_n), \delta$$

we form the directed *conjunction graph*, $G = (V, E)$, as follows:

- $V = \overline{X}_1 \cup \ldots \cup \overline{X}_n$[7] $\cup$ *Const* (*Const* is the set of constants appearing in the rule).
- $(v_1, v_2, \geq) \in E$ and $(v_2, v_1, \geq) \in E$ if $v_1 = v_2 \in \delta$
- $(v_1, v_2, \geq) \in E$ if $v_1 \geq v_2 \in \delta$ or $v_2 \leq v_1 \in \delta$
- $(v_1, v_2, >) \in E$ if $v_1 > v_2 \in \delta$ or $v_2 < v_1 \in \delta$
- $(v_1, v_2, \neq) \in E$ if $v_1 \neq v_2 \in \delta$
- $(v_1, v_2, \alpha) \in E$ if v_1 and v_2 are in *Const* such that $v_1 \alpha\ v_2$ is true. These are termed the *implicit edges* and will only be $>$ edges.

Note that there is no mention of the variables from the head; instead we call all those nodes equated to head variables *target nodes* and label them with a $*$ in the graphs. We also distinguish two types of edges: those between variable nodes are called *join* edges while the others are *restriction* edges. The join edges can be divided into *equijoin* edges if they are labeled with a $=$ and *non-equijoin* edges otherwise. Finally, from now on we will only talk about comparison operators involving the operators $>$, $\geq$, or $\neq$ since the others can be easily converted to these types. □

Example 3.3.2 (cont'd) For our example we obtain the graph shown in Figure 3.3. □

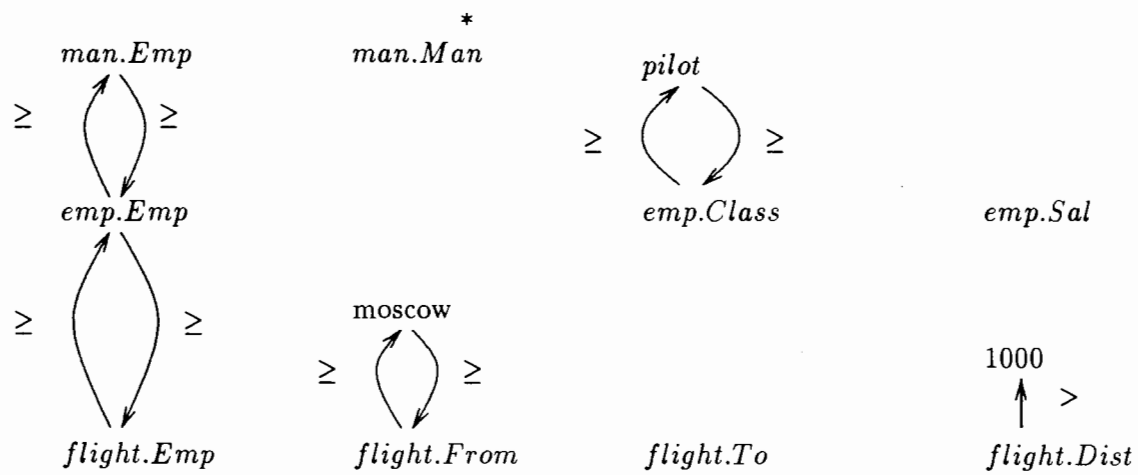


Figure 3.3: Conjunction Graph

[7] $\overline{X}_i$ is interpreted as a set of variables.

Some Definitions

In order to form the semantic expansion of a graph and later on to prove correctness, we need the notions of implication of edges and equivalence of graphs. The following definitions do not appear in [SO87, SO89] but do make explicit the underlying concepts presented in these papers.

Definition 3.3.2 The *dominant label* of a set of edges in G is either (a) the label of any edge in the set, if the labels of all the edges are the same, or (b) $>$ if there are two or more edges in the set with different labels.

The dominant label of a path of length one is the label of the edge in the path. While the dominant label of a path of length more than one is the dominant label of the set of edges in the path, provided the path contains no $\neq$ edges. If the path contains $\neq$ edges then the dominant label is undefined. □

Intuitively this tells us that if there is a path between two nodes x and y and the dominant label of the edges in the paths is α, then we can say that $x\ \alpha\ y$ is implied by the graph.

Definition 3.3.3 The graph G *syntactically implies* the edge $e = (x, y, \alpha)$, written $G \vdash e$, if there is path from the node x to the node y in G (or from y to x if α is $\neq$) such that α is implied by the dominant label of the path. □

For example, looking at the graph in Figure 3.3, we see that the edge $(flight.Dist, 1000, \neq)$ is implied by the graph.

The model theoretic view of implication is by assigning constants to variables:

Definition 3.3.4 An assignment of values (the mapping $m(\ )$) to the variables of a graph G *satisfies* G if for every edge (x, y, α) the comparison $m(x)\ \alpha\ m(y)$ is true. □

Definition 3.3.5 The graph G *implies* the edge e, written $G \models e$, iff for every assignment of values to variables which satisfies G, e is satisfied also. □

Definition 3.3.6 Two graphs, G and G', are *equivalent*, written as $G \equiv G'$, if every assignment that satisfies G satisfies G' and vice versa. This is similar to Definition 2.2.8 in Chapter 2. □

Theorem 3.3.1 below shows that $G \vdash e \Rightarrow G \models e$.

Condensed Canonical Form

This transformation is aimed at converting a conjunction graph G into a canonical form such that any other conjunction graph equivalent to G will have the same canonical form. We first condense the nodes of the graph into equivalence classes, and then minimise the number of edges.

Definition 3.3.7 We say $v \equiv v'$ for the nodes v and v' iff there is a cycle of $\geq$ edges involving v and v'. We can call this cycle the *equating cycle* and the $\geq$ edges involved the *equating edges* of v and v'. Using this, we define equivalence classes $|v| = \{v' \mid v \equiv v'\}$. □

Definition 3.3.8 For two sets of nodes x and y we define $E_{x,y}$ to be the set of all edges between them. □

Definition 3.3.9 The *condensed canonical* form of a graph G is $|G| = (V', E')$ where:

- $V' = \{|v| : v \in V\}$. V' is the partition of V with respect to $\equiv$ and the elements of V' represent sets of variables and constants. If any element of a node is a target attribute then the whole node is.
- E' is made up as follows:
  - $(|x|, |y|, \alpha) \in E'$ if the dominant label of the set $E_{|x|,|y|}$ is α.
  - An edge $(|x|, |y|, \alpha)$, where α is $>$ or $\geq$, is not in E' if it is the case that $|G| - \{(|x|, |y|, \alpha)\} \vdash (|x|, |y|, \alpha)$. We are thus forming the *transitive reduction* of $|G|$ [Meh84].

However, if either (a) there is an edge $(x, y, >)$ and $G \vdash y \geq x$ (the latter could occur if $x \equiv y$), or (b) $x \in |y|$ and there is an edge $(x, y, \neq)$, then $|G|$ is the null graph. In other words, G is contradictory. The above two cases are the only ways that this can happen [SO89].

Each node in the condensed graph represents a set of variables and constants, and since we want to refer to a condensed nodes, we pick a representative from the set as the name of the node (preferably a variable). □

The proof that $G \equiv |G|$ can be found in Theorem 3.3.2. The definitions in the previous subsection need to be adjusted slightly to take into account the fact that nodes can represent sets of variables and constants. For instance, to determine if $|G| = (V, E) \vdash (a, b, \alpha)$,

where a is not an element of V, but is in the set of variables and constants of the rule we are optimising, we will have to look at $|a|$ which will be in V.

Example 3.3.2 (cont'd) Figure 3.4 shows what happens when we form the canonical condensed form of the conjunction graph shown in Figure 3.3. The broken boxes indicate the equivalence classes.

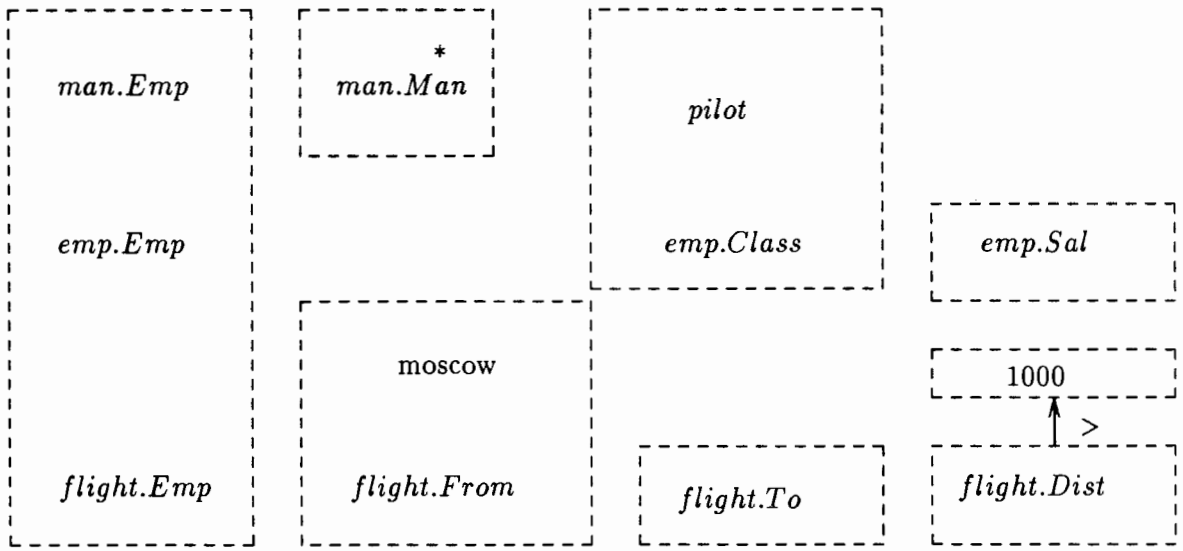


Figure 3.4: Condensed Graph

□

Example 3.3.3 Figure 3.5 shows the various stages when we form the canonical condensed form of a graph. The labels of the nodes are not shown. The graph on the left is the one we start off with, which is then transformed into the one on the right when we condense the nodes. Finally, we get the graph shown on the lower right when the transitive reduction of the edges is formed. □

Semantic Expansion

We now add those edges to the graph implied by the integrity constraints. The integrity constraints must be of the form $(A_1, \ldots, A_k \rightarrow)$, where each A_i is a comparison statement, which enables us to form the negation of any A_i as explained in Section 2.1.2.

Definition 3.3.10 We say that G syntactically implies an edge e with respect to a set of constraints I, written $G \vdash_I e$, if there is an integrity constraint $(A_1, \ldots, A_k \rightarrow)$in I such

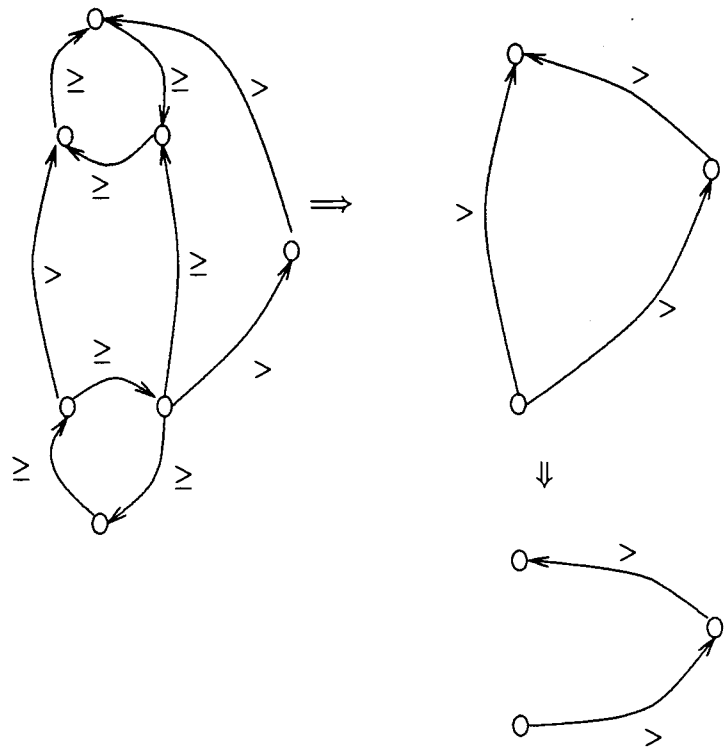


Figure 3.5: Condensed Graph

that $e = \neg A_j$ for some j and for all i such that $1 \leq i \leq k(i \neq j)$, we have $G \vdash A_i$. We call the edge e an *implied edge*.

Note that if there is an integrity constraint $(A_1, \ldots, A_k \rightarrow)$such that for every i we have $G \vdash A_i$, then we say that G is contradicted by the constraint set I and write $G \vdash_I$, and furthermore that G violates the integrity constraint.

We define $\models_I$ (implication with respect to the constraint set I) in similar way to $\models$. Every assignment which satisfies G *and* I satisfies e also. Equivalence with respect to the constraint set I, $\equiv_I$, is defined likewise. □

Definition 3.3.11 The semantic expansion of a graph $G = (V, E)$ with respect to a set of integrity constraints I is the graph $G^* = (V', E')$ where:

- For every $e \in E'$, either $e \in E$ or $G \vdash_I e$.
- V' is the set of vertices induced by E'.
- G^* is in canonical condensed form.

However if $G \vdash_I$ then G^* will be the null graph □

One can view the forming of the semantic expansion of a graph as a type of semantic closure operator. We get the null graph if the graph violates an integrity constraint and this means we can reject the rule.

Example 3.3.2 (cont'd) Figure 3.6 shows what happens when we form the semantic expansion of the condensed graph of Figure 3.5, using the residues

$$flight.From = moscow, flight.Dist \leq 3000 \rightarrow$$
$$emp.Class = pilot, emp.Emp = flight.Emp, flight.Dist > 2000, emp.Sal \leq 50 \rightarrow$$
$$emp.Class \neq pilot, emp.Emp = flight.Emp \rightarrow$$

Using the first residue, the system will add the edge $(flight.Dist, 3000, >)$. It will then add the implicit edge $(3000, 1000, >)$ and, when the transitive reduction is carried out, remove the edge $(flight.Dist, 1000, >)$. From the second constraint the system adds the edge $(emp.Sal, 50000, >)$.

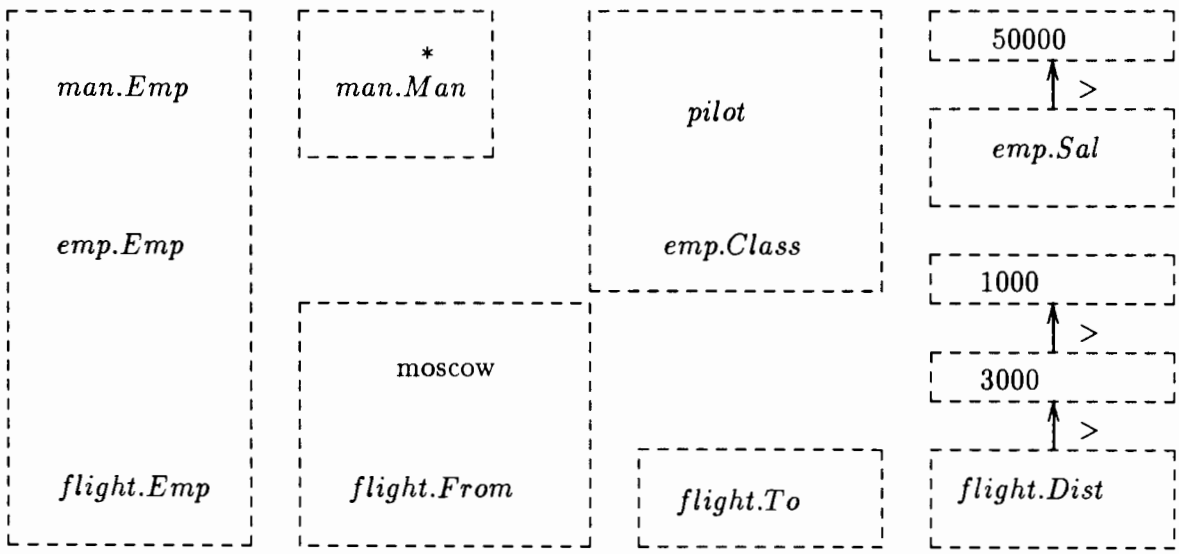


Figure 3.6: Semantic Expansion

□

Redundant Predicates

The aim of this transformation is to remove redundant predicates which in turn will lead to the removal of redundant vertices. We say that a predicate appears in a conjunction graph if it has any attributes that are nodes in the graph.

Definition 3.3.12 A predicate occurrence, R_i, appearing in a graph is *dangling* if all the following hold

- Any target node which contains an attribute of R_i also contains other attributes, or constants, not from R_i.
- There is at most one attribute in R_i that is a member of a node that has more than one element. In other words, there is at most one attribute in R_i connected to another attribute via an equi-join edge. If there is such an attribute A there must be a B and a subset constraint $S.B \subseteq R.A$ such that $A \equiv B$.
- There are no non-equijoin or restriction edges involving attributes from R_i.

□

The first two conditions ensure that the relation occurrence is adding nothing new to the answer, while the third ensures that the relation is not restricting the answer more than the other relation occurrences.

Definition 3.3.13 An occurrence, R_i, of a relation R is *duplicated* if there is another occurrence of R, R_j, such that for every attribute A of R we have $R_i.A \equiv R_j.A$. □

Definition 3.3.14 A graph is *redundant predicate free* if there are no dangling relations and no duplicated relation occurrences. □

Redundant Edges

Since join edges represent joins between relations the removal of these edges would be an advantage. On the other hand, some restriction edges are useful and some are not. The test cases in Section 5.4 give some examples of programs where a restriction is useful and some where it is not. Thus it is not wise to remove all redundant edges.

Definition 3.3.15 A graph G is *redundant edge free* if there is no $e \in E$ such that $G - \{e\} \vdash_I e$ and e is not profitable. That is we will not drop implied profitable edges. The profitability of an edge depends on operational considerations such as whether or not one of the nodes on the edge is indexed. □

Example 3.3.2 (cont'd) Figure 3.7 shows what the example graph looks like after the removal of redundant predicates and edges. The restriction edges $(flight.Dist, 3000, >)$ and $(emp.Sal, 50000, >)$ are implied by the constraints and so are removed. The implicit edge $(3000, 1000, >)$ is redundant since the constants 1000 and 3000 are not needed any more. The node containing *pilot* and *emp.Class* can also be removed since this is implied by the constraints. Finally, occurrences of the *emp* predicate are removed, since it is dangling, and non-target nodes not connected to any other node are removed.

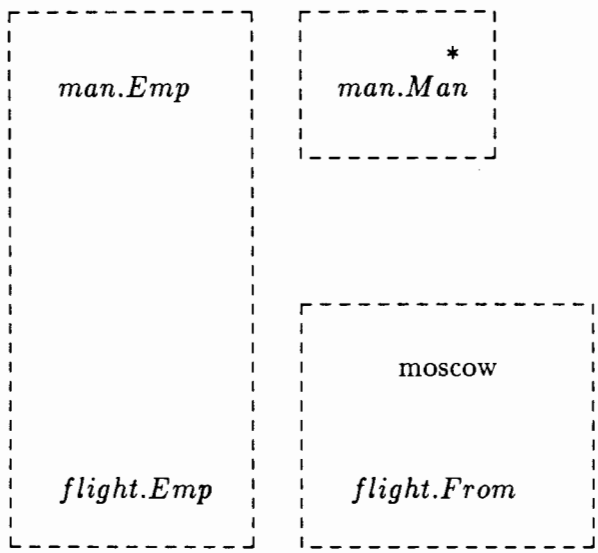


Figure 3.7: Redundancy Removal

□

Expanding Conjunction Graphs Back to Rule Form

The final transformation is to translate a conjunction graph back into Horn clause form.

Definition 3.3.16 For the graph $G = (V, E)$ we obtain the Horn clause:

$$p(\overline{X}) \ :- \ q_1(\overline{X}_1), \ldots, q_n(\overline{X}_n), \delta'$$

where $p(\overline{X}), q_1(\overline{X}_1), \ldots, q_n(\overline{X}_n)$ are from the original rule (except that we only retain those q_is that have attributes appearing on an edge) and δ' is built up as follows:

- We ignore implicit edges between constants (the system will have these built into it).

- "$x = y$" $\in \delta'$ if x and y are in the same node of G (i.e. $y \equiv x$) and $x = y$ can not already be inferred from δ'.
- "$x\ \alpha\ y$" $\in \delta'$ if $(|x|, |y|, \alpha) \in E$ and x and y are the representative attributes of the nodes $|x|$ and $|y|$ respectively.

□

Example 3.3.2 (cont'd) Our query finally reduces to

$$?- \quad man(?Man, Emp), flight(Emp, moscow, \_, \_).$$

Looking back at the original query, one can see that we have dropped one predicate occurrence (and therefore a join) and a comparison operator. □

3.3.2 Soundness

In this section, we aim to show that each of the above transformations does not alter the semantics of the graphs. The following proofs are original extensions of the short equivalence proof given in [SO87, SO89]. Chaining these proofs together and tagging on Theorem 3.2.2, which shows that residues follow logically from the constraints and rules, we will have shown that the optimised query (or rule) is semantically equivalent to the original query (or rule). The first theorem shows that $\vdash$ defined above is sound.

Theorem 3.3.1 *For every graph $G = (V, E)$ and edge $e = (a, b, \alpha)$, not necessarily in E but for which α is defined, we have that $G \vdash e$ implies $G \models e$.*

Proof: This is by induction on the length of the path between a and b. If the length is one then $e \in G$, so it automatically follows that if an assignment satisfies G then it satisfies e also.

Assume that the result holds for paths of length k or less and consider a path in G for e of length $k + 1$. We write the path for e as

$$a = v_0, e_0, \ldots, v_k, e_k, v_{k+1} = b$$

Let e' be (a, v_k, α') where α' is the dominant label of the path between a and v_k. Suppose we have an assignment which satisfies G, then it will also satisfy $e_k = (v_k, b, \alpha'')$ (α'' is the label of e_k) and, from the induction hypothesis, it will also satisfy e'. If α' and α'' are

the same, then from the transitivity of this operator (which cannot be $\neq$, as the definition of the dominant label of a path of length 2 or more excludes $\neq$), we get that (a, b, α) is satisfied also. If α' and α'' are different (neither can be $\neq$), α will be $>$ and by considering the different possibilities for α' and α'' one will see that $(a, b, >)$ is also satisfied. □

Theorem 3.3.2 *For every query graph, G, $G \equiv |G|$.*

Proof: Suppose we have an assignment which satisfies G. The assignment which satisfies G will also make the members of any cycle of $\geq$ edges true and so for each node on the cycle the same value will be assigned to them. Thus every variable in a node of $|G|$ will be assigned the same value. Each edge in $|G|$ is equal to, or implied by, an edge in G between a pair of nodes in two different $\geq$ cycles. So if the latter edge is made true by the assignment so will the former. Hence the assignment satisfies $|G|$ also.

The other way is slightly more difficult. Assume we have an assignment which satisfies $|G|$. All the variables and constants in a node of $|G|$ will be assigned the same value so all the equating edges in G will be true. Assume that $e = (|a|, |b|, \alpha)$ is an edge in $|G|$ made true by the assignment. We look at all the edges, in G, between elements of $|a|$ and elements of $|b|$ (i.e. $E_{|a|,|b|}$). If α is the same as all the labels on these edges then, since all the elements of $|a|$ have the same value and all elements of $|b|$ have same value, all the edges in $E_{|a|,|b|}$ are true also. If the labels on the edges in G are different, then α must be $>$. However, any assignment of values to $|a|$ and $|b|$ that makes $|a| > |b|$ true will also make $|a| \neq |b|$ and $|a| \geq |b|$ true, so all the edges in $E_{|a|,|b|}$ will be true. We also have to consider edges in G between elements of nodes in $|G|$ which are not mapped onto any edge in $|G|$, as they are transitively redundant. Since each operator (which will only be $>$ or $\geq$) we are considering has the transitivity property, a transitively redundant edge will be true if the edges on the path that implies its redundancy are true. □

Lemma 3.3.1 *Let G and G' be two query graphs and I a set of integrity constraints which G and G' do not violate. If G and G' are identical except that G' contains an edge which is the negation of some A_j in the body of an integrity constraint, $(A_1, \ldots, A_k \rightarrow)$, from I, and $G \vdash A_i$ for every $i \neq j$, then $G \equiv_I G'$.*

Proof: Suppose M is an assignment that satisfies G and I. Since M satisfies each element of I it must satisfy $(A_1, \ldots, A_k \rightarrow)$, in which case at least one of the A_is must be false under M. We know that $G \vdash A_i$ for every $i \neq j$, so from Theorem 3.3.1 we have that $G \models A_i$ for every $i \neq j$, so M must satisfy all A_i for $i \neq j$. Therefore it is only A_j that is made false by M. So $\neg A_j$ is satisfied by M and thus G with $\neg A_j$ is also satisfied by M. The latter is in fact G'. The other way is easier: if M satisfies G' then it automatically satisfies G since $G \subseteq G'$. □

Theorem 3.3.3 *Given a condensed query graph, G, and a set of integrity constraints, I, which every database applied to G satisfies, then the semantic expansion of G, G^*, is semantically equivalent to G.*

Proof: Since the addition of an implied edge preserves equivalence, the addition of a series of such edges will also preserve equivalence since $\equiv$ is transitive. □

Theorem 3.3.4 *Let G be a query graph with a dangling relation, R, and G' be a query graph identical to G except that the dangling relation and all it's associated nodes are missing. Furthermore, let I be a set of integrity constraints which includes the subset constraint $R.A \subseteq S.B$ which is the constraint used to determine that R is dangling. Then $G \equiv_I G'$.*

Proof: The definition of a dangling relation ensures that G' has the same as edges as G. So any assignment which satisfies one will satisfy the other. □

Theorem 3.3.5 *Let G be a query graph which has redundant edges with respect to a set of integrity constraints I, and let G' be a query graph identical to G except that these edges have been removed, then $G \equiv_I G'$.*

Proof: From Lemma 3.3.1 we see that the addition of a series of implied edges preserves equivalence thus the removal of a series of implied edges will also preserve equivalence. □

Further support for the conjecture that the transformations preserve semantic equivalence can be found in [SO89].

3.4 Conclusion

In this chapter, I have introduced semantic optimisation, semantic knowledge and described two optimisation systems in detail. These will be dovetailed together to form part of the DatalogIC semantic optimisation system. The first of these systems, CGM, takes a logical view of semantic optimisation and most probably subsumes many previous systems. However, although a sound algorithm is given for the first part of the scheme (residue calculation), it is not clear from [CGM88] how one uses these residues (although [CGM90] does give some higher-level ideas). The second system, SO, is designed from the database view and as such does not take into account either relations defined by rules or Horn clause constraints, but does show how one can apply residues to rules. Each system fills in the gaps of the other leading to a system which can be implemented, and this is dealt with in more detail in Section 4.5.2. It must be pointed out that CGM is the stronger of the two partners and SO can be viewed as a lower level, but more implementable, view of the last stage of CGM. The next chapter describes the complete semantic optimisation system for the DatalogIC language, drawing on ideas from this chapter.

Chapter 4

DatalogIC and its Semantic Optimisation

In this chapter, I draw together the ideas introduced in previous chapters on semantic optimisation and semantic knowledge. I synthesise a full semantic optimisation system for Datalog programs, parts of which I have implemented. In Section 4.1, I outline the language DatalogIC which has been devised to serve as a vehicle for semantic optimisation techniques. A short sample session on the DatalogIC system is then given in Section 4.2. In Sections 4.3 and 4.4, I describe how semantic knowledge can be managed and give some ideas on the optimisation of recursive programs. Finally, in Section 4.5, I describe a system for the semantic optimisation of DatalogIC programs, viewing the task both globally and locally. The local optimiser, which is a fusion of the CGM and SO systems introduced in the previous chapter, has been implemented. The remainder of what I describe below has not been fully implemented, and can be seen as a design proposal. Only the simplest global optimiser and constraint manager have been implemented and serve as stubs for the local optimisation system. The recursive component optimiser has not been implemented at all. The next chapter gives more details of the implementation itself.

4.1 The DatalogIC Language

The language that I have devised in order to write programs and express semantic knowledge is, basically, a more structured version of Datalog that can take semantic knowledge in Horn clause form as part of the program. An attempt has also been made to introduce

some conciseness into the language.

The added structure is achieved through the *predicate definition*; one definition for each predicate in the program. The aim of this construct is to introduce some modularisation into Datalog programs. There are two types of predicate definition: one for intensional predicates, the other for extensional predicates. A definition for an intensional predicate starts with the `INT` reserved word, which is followed by the name of the predicate and its argument list (which must all be variables). This is the head of the definition. The definition contains the rules which define the predicate and the integrity constraints which describe it. The syntax for rules and integrity constraints is the same as that described in Section 2.1. Any rule in the definition which has a head matching the definition (same head and arguments) can be written with the head omitted. Definitions for extensional predicates start with the `EXT` reserved word, followed by the definition head. Extensional definitions contain integrity constraints, as well as information about the underlying database relation this predicate is mapped onto. This information specifies what the indices for the relation are, and other operational knowledge [1]. The head of extensional definitions must contain the name of the relation being described and its attributes as they appear in the database. If this was not done, we could not convert the rules into SQL expressions. The name of the relation and the arguments in an intensional definition are also used as the name for the SQL view definition for the predicate. Section 5.3 gives more details on this.

Since the integrity constraints inside both types of definition are about the predicate being defined, the parser will automatically place an occurrence of the definition head inside the body of each constraint. This feature, and the one which allows rule heads to be omitted, provides a type of "context-sensitive shorthand" and adds some conciseness to the language. Example 4.1.1 illustrates what this looks like in a DatalogIC program. It is possible to write rules and integrity constraints in the normal way by not placing them in any definition. Thus DatalogIC is a superset of function-free Datalog.

As well as a set of predicate definitions, a program contains a set of query forms, as defined in Definition 2.1.4. These specify what queries the program will accept. Variables in a query are preceded by a label. A "?" indicates that the variable is an input variable, while a "!" indicates that it is an output variable. At runtime, users are given a list of

[1] The writing of operational knowledge in extensional definitions has not been implemented.

query forms from which they select a query to be evaluated. The system collects bindings for the input variables in the query selected, processes the query and returns bindings for the output variables. The query forms are important as the system is made aware of the type of queries that are going to be asked and can therefore optimise and compile the program accordingly [2]. Clearly the user should be allowed to present arbitrary queries but this will be at the expense of optimal performance.

Before presenting an example program, I give the Backus-Naur specification of DatalogIC:

```
Program          ::= { Statement }

Statement        ::= Ext_Definition | Int_Definition | Rule |
                     Integrity_Constr | Query

Int_Definition   ::= 'INT' Head_Predicate '{' { Inside_statement } '}'

Ext_Definition   ::= 'EXT' Head_Predicate '{' { Integrity_Constr  } '}'

Inside_statement ::= Headless_Rule | Rule | Integrity_Constr

Headless_Rule    ::= ':-'  Horn_Cl_Body '.'

Rule             ::= Predicate  ':-'  Horn_Cl_Body '.'

Query            ::= [Predicate] '?-'  Horn_Cl_Body '.'

Integrity_Constr ::= 'IC' Horn_Cl_Body '->' '.' | 'IC' -> Predicate '.' |
                     'IC' Horn_Cl_Body '->' Predicate '.'

Horn_Cl_Body     ::= { Predicate | Comparison_Op  }

Predicate        ::= PRED_ID '(' Argument  { ',' Argument } ')'
```

[2]The querying of the user for values for the input variables has not been implemented.

```
Head_Predicate   ::= PRED_ID '(' Variable  { ',' Variable } ')'

Comparison_Op    ::= Argument OPERATOR Argument

Argument         ::= Variable | Constant

Variable         ::= [ Label ] VAR_ID

Label            ::=  '!' | '?'

Constant         ::= STRING | INTEGER
```

The lexical symbols are as follows:

- PRED_ID is an alphanumeric string beginning with a lowercase letter.
- VAR_ID is an alphanumeric string beginning with a capital letter or the "don't care" variable "_".
- STRING is an alphanumeric string delimited with single quotes.
- OPERATOR is one of $\{>, >=, <, <=, !=, =\}$.
- Comments are delimited with /* and */.
- // indicates that the rest of the line is a comment.

Note that only query variables have labels.

Example 4.1.1 Here is a sample DatalogIC program as it would appear in a text file:

```
EXT emp(EName,Edept,Sal){
        // All employees earn more than 5000.
        IC -> Sal > 5000.
        // A manager of a department earns more than the members of
        // the department.
        IC manager(Mname,Edept,S) -> S > Sal .
```

```
}

EXT manager(Mname,Dept,Sal){
}

// This gives us managers earning less than 4000, and their employees.
INT lowmanager(X,Y){
        :- manager(X,Dept,S), emp(Y,Dept,S2), S < 4000.
}

?- lowmanager(!Mname,?Ename).
```

The rules and constraints in this program are similar to those in Example 3.1.4. The program contains two extensional predicate definitions and one intensional predicate definition. The definition for the relation *emp* contains two constraints, written in the shorthand form mentioned. Since they appear in the definition for *emp*, the parser will add an occurrence of the head to the body of each. For example, the full form of the first constraint is

$$emp(Ename, Edept, Sal), Edept = admin \rightarrow Sal > 5000.$$

The second constraint can be expanded likewise. If the definition was empty it would still serve a purpose, as the the head would be used by the rule-to-SQL converter to determine the name of the relation the predicate is mapped onto, as well its column names. In the definition for *lowmanager*, there is a rule written in the short form which in its full form would have the definition head appended to the front of it. The final line of the program is the query. The variable *Ename* is the input variable and *Mname* is the output variable. The query is thus short for: "Give me the names of all employees whose managers earn less than 4000". It is possible to optimise this program as shown in Example 4.5.3. □

4.2 Sample Session

In this section I show, from the users point of view, how DatalogIC programs are parsed and compiled, and how queries are evaluated. Chapter 5 deals with the implementation of

these processes. Programs are entered by the user into a normal ASCII text file. The user then runs the DatalogIC system and types the **parse** command. The program is parsed and any errors found are reported. The program is then ready to be compiled into SQL statements. The SQL statements are either written to a file or passed onto Oracle (in which case the program will connect itself to Oracle), depending on an option set by the user. After the program has been compiled, the user can ask the system to run any of the query forms in the program. Prior to compilation, the user has the option of submitting the program to the semantic optimiser (or Magic Sets optimiser) by entering the **optimise** command (or **magic** command).

Example 4.2.1 Below is a simple, sample DatalogIC session where a recursive program (the transitive closure program of Example 2.1.1) is parsed, compiled and queried. System input and output is in **typewriter font** while comments are in *italics*. For more details on these user commands available see Section 5.2. A sample session showing the optimiser in action is given in Section 4.5.2, while in Section 5.4 some more examples and the times they took to run are given. □

Starting at the UNIX prompt $ *the user enters the DatalogIC system...*

```
$ dlog

Welcome to DatalogIC. Enter '?' for help.

dlog> parse rt

Program rt has been parsed. (0 errors 0 warnings).
Log file is rt.lis

dlog> list program

INT path ( X00, Y01){
        path ( X00, Y01)  :-  link (  X10, Y11 ),
Y01 =  Y11,  X00 =  X10,
        path ( X00, Y01)  :-  path (  X10, Z11 ),  link (  Z20, Y21 ),
Y01 =  Y21,  Z11 =  Z20,  X00 =  X10,
}

EXT link ( X00, Y01) {
        File name : link Index : None
}
```

```
Integrity Constraints
None

Query0(?X,!Y) :- path(X,Y), Y = 'A(10,0)'.
```

Note that the program is fully rectified

```
dlog> compile

Compiling Queries. SQL statements sent to rt.sql.
Connected to Oracle user: mark
```

The link relation contains tuples forming a binary tree of depth 10 (see Section 5.4 for more details of how this is done).

The following table show the times (in seconds) taken for the various stages of the processing of this program. The first is the time taken for the system to set up the tables and views for Semi-Naive evaluation, the second is the time spent running Semi-Naive and the third is total time spent compiling the program.

```
+-------------------+-----------------+
| Activity          | Time (secs)     |
+-------------------+-----------------+
| SN Set Up         |  0:0:27 (27)    |
```

Times for Semi-Naive iterations have been omitted.

```
| Semi-Naive to run |  0:1:3 (63)     |
| Compilation       |  0:3:26 (206)   |
+-------------------+-----------------+
| Total Time        |  0:10:11 (611)  |
+-------------------+-----------------+

dlog> query Query0
Query output sent to rt.sqlout

10 row(s) selected
Query took  0:0:1 (1) seconds

dlog> quit
```

The query has selected those tuples that link A(10,0) to all of its ancestors. These are written to the file rt.sqlout which looks as follows.

```
Result of query: Query0
-------------------------
```

X	Y
A(0,0)	A(10,0)
A(1,0)	A(10,0)
A(2,0)	A(10,0)
A(3,0)	A(10,0)
A(4,0)	A(10,0)
A(5,0)	A(10,0)
A(6,0)	A(10,0)
A(7,0)	A(10,0)
A(8,0)	A(10,0)
A(9,0)	A(10,0)

□

4.3 Management of Semantic Knowledge

Semantic knowledge was introduced in Chapter 3, and in Section 4.5 I will describe how it is used. In this section, I present some ideas on how it can be managed. Any practical implementation, which carries out semantic optimisation, must not only attempt optimisation of recursive programs, but must also manage semantic knowledge effectively. A management policy should include consistency and redundancy checking, inference of new semantic knowledge and controlled access to the semantic knowledge by the optimisation algorithm; these are all features of a knowledge base. In this section, I first outline the required functionality of the constraint manager, and then give four rules which form the core of a simple constraint manager. I assume that the semantic knowledge is represented as a set of Horn clauses and take a general proof-theoretic approach. However, it may be possible to use some other representation along with a more sophisticated system. Other researchers [Kin81, Jar86, MZ87] have mentioned that the semantic knowledge should be held in some sort of knowledge base, but have omitted to describe it. However in [SO89] management of semantic knowledge is discussed and I use this as the starting point for my ideas. Only the part of the manager that packages constraints for the optimiser has been implemented.

4.3.1 Functions of a Semantic Knowledge Manager

Interface

The manager should have a well-defined interface with the optimiser, the rule/goal graph manager and the program parser. Integrity constraints are passed by the parser to the constraint manager, which in turn passes these to the optimiser when requested to do so. The optimiser will then use the constraints to calculate residues for a rule. One form that the interaction could take is the following [3]: The optimiser calls the constraint access function, passing it the rule being optimised; the manager then builds a list of integrity constraints that could possibly be merge compatible with the rule (which happens if a predicate has an occurrence in the body of the rule and in the integrity constraint); this list is then ordered (in ascending order) on the number of predicates with an occurrence in the constraint but not in the rule. If all the predicates in a constraint appear in a rule, then the constraint will have a zero score, be at the top of the list, and be used first by the residue calculator. The advantage of this is obvious: the smaller the number of literals in the constraint that do not appear in the rule, the fewer conjuncts there will be in the residue, which implies that it will have more of an effect on the rule [4]. The manager also requires access to the rule/goal graph as part of its constraint inference function.

In the same way that users are not expected to enter optimal declarative programs, they should not be expected to enter semantic knowledge which is optimal. The knowledge entered by the user might contain redundancies, might not contain all the useful information explicitly and might even be inconsistent. In the following, I consider each of these three aspects. Figure 4.1 shows the structure of a constraint manager which carries out the above functions.

Consistency and Redundancy

Consistency is the easiest notion to define: if we can prove both a fact and its negation then the constraint set is inconsistent. What we should do when we find that the constraint set is inconsistent is a harder question to answer. The easiest, but by no means the most

[3] This is what happens in the DatalogIC system as it is currently implemented.

[4] The true relationship between the number of predicates in an IC that are not in a rule, i, and the number of predicates in the residue, j, is $j \geq i$ since there may be an occurrence of a predicate in the IC that does not unify with an occurrence of the same predicate in the rule.

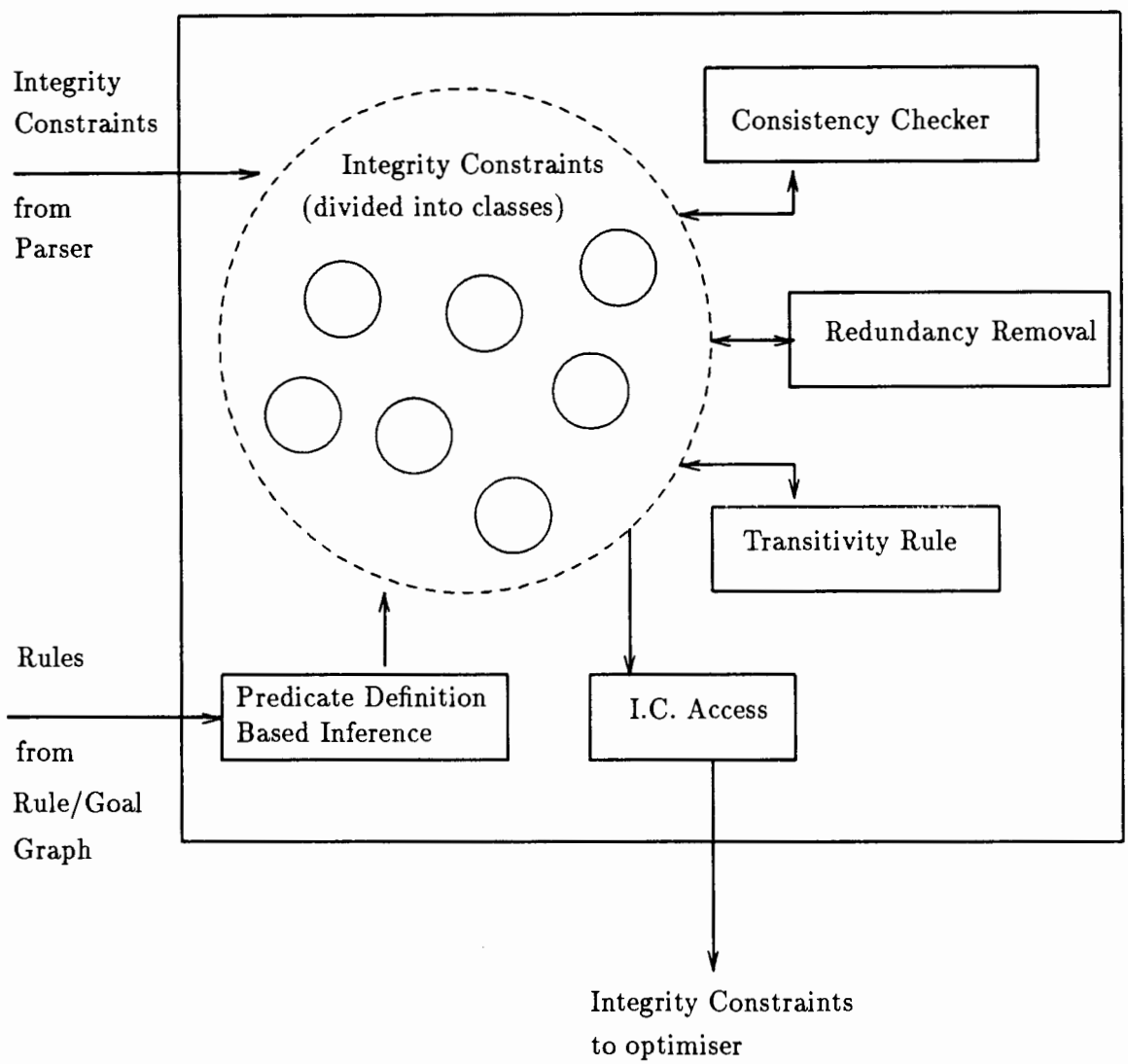


Figure 4.1: Structure of an Integrity Constraint Manager

user-friendly approach, is to reject the knowledge base in its entirety. A better solution is given at the end of Section 4.3.2 . Redundancy is a more difficult notion to define since it depends on external considerations, such as how the constraints are used. It is not good enough to say that any fact that is inferable from the knowledge base, less the fact itself, is redundant, since this could lead to a constraint that was generated during the inference phase being removed during redundancy removal. So we have to add the extra condition that, if the fact is not usable by the system then it will be redundant.

Example 4.3.1 Suppose we have the following two constraints

$$\begin{aligned} p(X,Y) &\rightarrow X > 40. \\ p(X,Y), q(X,Y) &\rightarrow X > 20. \end{aligned}$$

If the second constraint merges with any rule to produce the residue ($\rightarrow X > 20$), then clearly the first constraint will also merge with the same rule to produce the residue ($\rightarrow X > 40$), which implies the second residue. Therefore the second constraint is of no greater use than the first and, since the second is logically implied by the first, it can be labeled as redundant. □

A formulation of the notions of consistency and redundancy is given later on in Section 4.3.2.

Inference

The optimiser does not have the ability to use knowledge in the constraint set that is implicit, so if a constraint is to be used by the optimiser it must be made explicit. Inference of semantic knowledge is the method by which knowledge implicit in the constraints contained in the DatalogIC program can be revealed. The inference of new semantic knowledge can either be done using the constraints only, or in combination with the rules of the program. The former has been done using a proof system, as given in [SO89], and is detailed in Section 4.3.2. In Example 4.3.1, the second constraint is implied by the first with aid of an inference rule, the augmentation rule, which allows the arbitrary addition of literals to the body of a constraint. Inference using the rules of a Datalog program is, to my knowledge, something that has not been presented before. However, it has been done with dependencies and relation views [KP82]. The rationale behind such inference is that if we have a set of rules defining a relation, and we have some semantic knowledge about

the *defining* relations, then it should be possible to infer some new knowledge about the *defined* relation. The nature of this inference will depend on the defining rules.

Example 4.3.2 Suppose we have the following set of rules

$$\begin{aligned}
&(1)\quad p(X,Y) \;:-\; r(X,Z).\\
&(2)\quad p(X,Y) \;:-\; s(X,Z,W), t(W,Y).\\
&(3)\quad t(X,Y) \;:-\; p(X,Z), q(X,Y).\\
&(4)\quad q(X,Y) \;:-\; u(X,Y).
\end{aligned}$$

and the set of constraints:

$$\begin{aligned}
&(A)\qquad r(X,Y) \;\rightarrow\; X < 10.\\
&(B)\quad s(X,Z,W) \;\rightarrow\; X < 20.
\end{aligned}$$

Constraint A says that the first argument of any tuple from $r(X,Y)$ will be less than 10. Constraint B says that the first argument of any tuple from $s(X,Z,W)$ will be less than 20. From the first two rules we see that the first argument of any tuple from $p(X,Y)$ will either have come from $r(X,Y)$, in which case it will be less than 10, or have come from $s(X,Z,W)$ in which case it will be less than 20. One can therefore conclude that the first argument of a tuple from $p(X,Y)$ will always be less than 20. In other words we get the new constraint

$$(C)\quad p(X,Y) \;\rightarrow\; X < 20.$$

□

The above example shows how we can pass semantic knowledge up the rule/goal graph. It is also possible to pass it across and down the tree, which is akin to the pushing of select operators inwards in relational algebra expressions and is similar to the work presented in [KL85].

Example 4.3.2 (cont'd) From constraint C and rule 3 we can see that any tuple from $q(X,Y)$ used in the proof of a tuple in $t(X,Y)$ will have to have $X < 20$. Thus the only tuples from q that are going to be used in the above program are going to be a subset of q where $X < 20$. This is in effect semantic knowledge passed *across* the tree. However, it is of a different type to the previous inference since we cannot add the constraint

$$q(X,Y) \rightarrow X < 20$$

This is because q could contain tuples with $X \geq 20$ without contradicting the initial constraints A and B.

In a similar way we can infer that the only tuples from u that are going to be used in the program will have $X < 20$. This is semantic knowledge passed *down* the tree. Again, we cannot add the constraint

$$u(X,Y) \rightarrow X < 20$$

but we can add the rule

$$u'(X,Y) \ :- \ u(X,Y), X < 20.$$

and replace all occurrences of u by u'. This is the same as if we had pushed the constraint down the tree. □

In the next subsection I present the four rules on which the constraint manager for DatalogIC is based.

4.3.2 Proof System

We will consider only non-recursive programs (or the non-recursive part of recursive programs) and only constraints which can be rewritten in terms of EDB predicates. The constraints must also be of the form $(A_1, \ldots, A_n \rightarrow)$ and each must have unique variables. In order to prevent the exclusion of constraints with non-evaluable heads we will allow at most one occurrence of a negated predicate in the body of the constraint. In [SO89] it is pointed out that the set of legal constraints is a subset of first order predicate calculus, and so we can use one of the natural deduction systems devised for first order logic. They present a proof system from which I have derived the first three rules of the system for DatalogIC. These are as follows:

1. If we can infer both $(A(\overline{X}) \rightarrow)$ and $(\neg A(\overline{X}) \rightarrow)$, then the system is inconsistent.

2. A constraint of the form $(A(\overline{X}), B(\overline{Y}) \rightarrow)$ is redundant if there is already a constraint of the form $(A(\overline{Z}) \rightarrow)$ in the constraint set and $A(\overline{X})$ unifies with $A(\overline{Z})$.

3. If there are constraints of the form $(A(\overline{X}), \neg B(\overline{Y}) \rightarrow)$ and $(B(\overline{Z}), \neg C(\overline{W}) \rightarrow)$ in the constraint set, then we can infer $(A(\overline{X}), \neg C(\overline{W}) \rightarrow)\theta$ where θ is the most general unifier of $B(\overline{Y})$ and $B(\overline{Z})$.

The transitivity inference rule, R_3, is useful as it makes a constraint applicable when, on its own, it is not. For instance, if we had a constraint $(B(\overline{Y}), \neg C(\overline{Z}) \rightarrow)$, where C is an evaluable predicate and $B(\overline{Y})$ does not appear in any rule, then this constraint is useless, as far as the local optimiser is concerned [5]. However, if we had a constraint $(A(\overline{X}), \neg B(\overline{Y}) \rightarrow)$, and $A(\overline{X})$ does appear in a rule, then by being able to infer $(A(\overline{X}), \neg C(\overline{Z}) \rightarrow)$ we have made $C(\overline{Z})$ available to the optimiser.

To infer semantic knowledge using the rules of the program, we use the rule R_4 which I have devised and is as follows.

Definition 4.3.1 If we have the following set of n rules defining a predicate p in our program

$$\{p(\overline{X}) :- A_i(\overline{Y}_i), Q_i(\overline{Z}_i) \mid 1 \leq i \leq n\}$$

and the set of integrity constraints

$$\{Q_i(\overline{W}_i), B_i(\overline{V}_i) \rightarrow \mid 1 \leq i \leq n\}$$

such that

1. A_i, Q_i and B_i are conjunctions of predicates,
2. $\overline{X}$, $\overline{Y}_i$, $\overline{Z}_i$, $\overline{W}_i$ and $\overline{V}_i$ are vectors of variables,
3. each constraint contains unique variables (different from each other and the rules), and
4. $Q_i(\overline{Z}_i)$ unifies with $Q_i(\overline{W}_i)$, the most general unifier being θ_i,

then we can infer the constraint

$$p(\overline{X}), (B_1(\overline{V}_1), \ldots, B_n(\overline{V}_n))\theta \rightarrow$$

where θ is the composition of all the θ_i's □

If some of the B_i's are evaluable predicates, it may be possible to simplify the body of the new constraint using the axioms for these predicates. For instance, we used this inference rule in Example 4.3.2. There $B_1(\overline{V}_1)$ was $X \geq 10$ and $B_2(\overline{V}_2)$ was $X \geq 20$, so we inferred

[5] As I point out later the local optimiser will only use constraints that give residues containing no non-evaluable predicates

$$p(X,Y), X \geq 10, X \geq 20 \quad \rightarrow .$$

and simplified this to

$$p(X,Y), X \geq 20 \quad \rightarrow .$$

The first three rules, R_1 to R_3, are standard inference rules, and are well-known to be sound. It remains to prove the new rule, R_4.

Theorem 4.3.1 *If it is the case that for every constraint I we can write I in terms of EDB predicates only (i.e. I^* exists), then R_4 is a valid inference rule.*

Proof: Assume we have a set of rules defining a predicate p along with a set of integrity constraints all conforming to those in Definition 4.3.1. Then to show that the rule is valid we show that from these antecedents we can reach the conclusion. Note that we will be using the $\vee$ introduction rule, which allows us to infer $(A \vee B \rightarrow)$ from the constraints $(A \rightarrow)$ and $(B \rightarrow)$.

$$\{Q_i(\overline{W}_i), B_i(\overline{V}_i) \rightarrow \mid 1 \leq i \leq n\}$$

Augmentation Rule

$$\{A_i(\overline{Y}_i), Q_i(\overline{W}_i), B_1(\overline{V}_1), \ldots, B_n(\overline{V}_n) \rightarrow \mid 1 \leq i \leq n\}$$

$\vee$ introduction.

$$A_1(\overline{Y}_1), Q_1(\overline{W}_1), B_1(\overline{V}_1), \ldots, B_n(\overline{V}_n) \vee \ldots \vee A_n(\overline{Y}_n), Q_n(\overline{W}_n), B_1(\overline{V}_n), \ldots, B_n(\overline{V}_n) \rightarrow$$

Distributive Rule and applying θ, which has no effect on Y_i and converts W_i to Z_i.

$$(A_1(\overline{Y}_1), Q_1(\overline{Z}_1), \vee \ldots \vee, A_n(\overline{Y}_n), Q_n(\overline{Z}_n)), (B_1(\overline{V}_1), \ldots, B_n(\overline{V}_n))\theta$$

Definition of p

$$p(\overline{X}), (B_1(\overline{V}_1), \ldots, B_n(\overline{V}_n))\theta \rightarrow$$

□

We now have at our disposal inference rules for infering constraints, either using constraints on their own, or using the rules which define predicates. Also, we have a way for checking consistency and redundancy. For each constraint there is a relevant subset of the constraint set which is the smallest set from which the constraint can be proved. [SO89] proposes that we divide the constraints into equivalence classes, which helps to isolate the relevant sets and to give structure to the set of constraints. A constraint is in

an equivalence class if it shares a predicate with at least one other member of the class. It is clear that when we are using the transitivity inference rule we need only look at constraints which are in the same equivalence class.

In summary, an algorithm that the constraint manager could use would be made up of the following procedures (the E_i's are the equivalence classes into which we have divided the constraint set) which correspond to the inference rules R_1 to R_4 :

1. If $(A \rightarrow) \in E_i$ and $(\neg A \rightarrow) \in E_i$, then E_i is inconsistent and should be dropped.
2. If $(A, B \rightarrow) \in E_i$ and $(A \rightarrow) \in E_i$, then remove $(A, B \rightarrow)$ from E_i.
3. If $(A, \neg B \rightarrow) \in E_i$ and $(B, \neg C \rightarrow) \in E_i$, then add $(A, \neg C) \rightarrow$ to E_i.
4. Add any constraint implied by R_4. This may lead to the merging of two equivalence classes.

The first three procedures would be executed for each equivalence class E_i while the fourth one would be run using all the union of all the classes.

In this section I have discussed a semantic knowledge manager, which is simple and has sound theoretical foundations. The implementation of the system is important, since resources consumed by the manager will count towards the cost of optimisation, which as I mentioned in Section 2.4, must itself be efficient. This is left for future work.

4.4 Optimisation of Recursive Programs

The evaluation of a recursive program can consume a lot of resources and, since recursion is an important modeling tool, this is going to be a bottleneck in the development of an efficient deductive database language. The problem is further compounded by the fact that it is difficult to estimate how long a given recursive program will take to evaluate. This is in contrast to non-recursive programs, where we merely count the number of joins in the compiled form of the query to get a time estimate. Comprehensive studies have been made of the various classes of recursive programs, and algorithms that are best suited to a particular class have been devised [Nau87, BR86]. Some classes of recursive program are cheaper to evaluate than others, so it is clearly beneficial if we can convert a recursive program into one that is equivalent to the original but can be evaluated more efficiently.

An extreme example of this inter-class movement is when a recursive program can be rewritten into an equivalent program that is actually not recursive. We then say that the original program exhibits *data independent* or *bounded recursion* [Nau86]. We can also have the case where a program is bounded on a particular condition; if certain variables are bound to constants that satisify the condition, then the recursion is bounded.

Very little work has been done on the semantic optimisation of recursive programs. [HL88] discusses the problem but restricts the evaluation algorithm to the Henschen-Naqvi compilation system [HN84]. In this section, I present some ideas on semantic optimisation of recursive programs, without being restricted to a particular evaluation system. None of the following has been implemented in the DatalogIC system.

Motivating Examples

It is important to distinguish between the optimisation of a recursive rule and the optimisation of a recursive component. The latter is the subject of this section, while the former can in fact be achieved by the local optimisation system discussed in Section 4.5.2, as the following example shows.

Example 4.4.1 We have the program

$$\begin{aligned} path(X,Y,Score) &:- link(X,Y,Score). \\ path(X,Y,Score) &:- path(X,Z,S), link(Z,Y,Score). \end{aligned}$$

and the constraint

$$path(A,B,S), link(B,C,Score) \rightarrow S = 1.$$

which says that tuples from *path* which can be joined with tuples from *link* have a score of 1. In order to satisfy this constraint we set the score of edges into non-sink nodes to 1. The optimiser will rewrite the second rule to

$$path(X,Y,Score) : -path(X,Z,S), link(Z,Y,Score), S = 1$$

which is in effect a scan reduction (see Section 5.4 for more details on what this entails, and the savings that are possible). When the program is evaluated, by Semi-Naive evaluation, the restriction will filter out those tuples from *path* which will not join with tuples from *link* before a search of *link* is made. This can save a considerable amount of time if *link* is large. □

Example 4.4.2 This is a good example of what the system I propose should be able to handle. Suppose we are given the program

$$\begin{aligned} path(X,Y) &:- link(X,Y). \\ path(X,Y) &:- path(X,Z), link(Z,Y). \end{aligned}$$

and the constraint on the *link* relation

$$link(A,B), link(B,C), link(C,D) \rightarrow$$

This constraint says that there are no paths of length three or more. The program, which attempts to find paths of any length, can therefore be rewritten as

$$\begin{aligned} path(X,Y) &:- link(X,Y). \\ path(X,Y) &:- link(X,Z), link(Z,Y). \end{aligned}$$

The new program will find paths of length one or two only, which is all that can be done. This is an example of *semantically bounded recursion*, since it relies on knowledge about the data in the database, i.e. the semantics of the database. This is the main type of optimisation the system I propose will carry out. □

The system should also be able to recognise those cases of data-independent recursion that have been investigated in the literature. In the next part of this section, I will give some ideas on how integrity constraints can be used to optimise recursive programs. First, I will give some useful foundational ideas and then directions for further research.

4.4.1 Proposed System

Some useful tools in the study of recursive programs are *expansion trees* and *expansion sets* [Nau87]. The expansion tree of a predicate is formed by repeatedly expanding the rules of the program, starting with the rules defining the predicate, in much the same way that the productions of a grammar can be expanded. Every predicate in a Datalog program has an expansion tree and from it we can form the predicate's expansion set, which is the set of terminal nodes of the expansion tree.

Definition 4.4.1 The *expansion tree* of a predicate $p(\overline{X})$ is a tree where each node is labeled with a conjunction of predicate occurrences [6]. The tree is built up as follows:

[6] From here on I shall not distinguish between the node of the tree and its label.

1. The root of the tree is $p(\overline{X})$.

2. A child of a node $p_1(\overline{X}_1), \ldots, p_n(\overline{X}_n)$ is

$$p_1(\overline{X}_1), \ldots, p_{i-1}(\overline{X}_{i-1}), (q_1(\overline{Z}_1), \ldots, q_m(\overline{Z}_m))\theta, p_{i+1}(\overline{X}_{i+1}), \ldots, p_n(\overline{X}_n)$$

if there is a rule in the program of the form

$$p_i(\overline{Z}) : -q_1(\overline{Z}_1), \ldots, q_m(\overline{Z}_m)$$

and the most general unifier of $p_i(\overline{X}_i)$ and $p_i(\overline{Z})$ is θ. We say that we have *applied* the rule to the node.

□

For EDB predicates the expansion tree is composed of one node only, and for recursive predicates the expansion tree is infinite. Furthermore, if a predicate $p(\overline{X})$ is used to define a predicate $q(\overline{Z})$, then the expansion tree for $p(\overline{X})$ can be used to build up the expansion tree for $q(\overline{Z})$.

The expansion set of a predicate is formed by taking all the terminal nodes of the expansion tree. These will be conjunctions of EDB predicates.

Example 4.4.3 The set of rules:

$$\begin{aligned} (1)\quad & p(X,Y) \;:-\; q(X,Y). \\ (2)\quad & p(X,Y) \;:-\; s(X,Y). \\ (3)\quad & p(X,Y) \;:-\; p(X,Z), s(Z,Y). \end{aligned}$$

has expansion tree shown in Figure 4.2. The expansion set is

$$\{q(X,Y), s(X,Y), q(X,Z_0)s(Z_0,Y), s(X,Z_0)s(Z_0,Y), \ldots\}$$

□

The expansion tree for a predicate $p(\overline{X})$ indicates how tuples in the relation p can be derived. For instance, from the terminal node $q(X,Z_0), s(Z_0,Y)$ in the above example we see that if there is a tuple (a,b) in the relation q and a tuple (b,c) in the relation s, then the tuple (a,c) is in p. The importance of the expansion set for a predicate is that it is the compiled form of the predicate. To calculate the relation for the predicate at the root

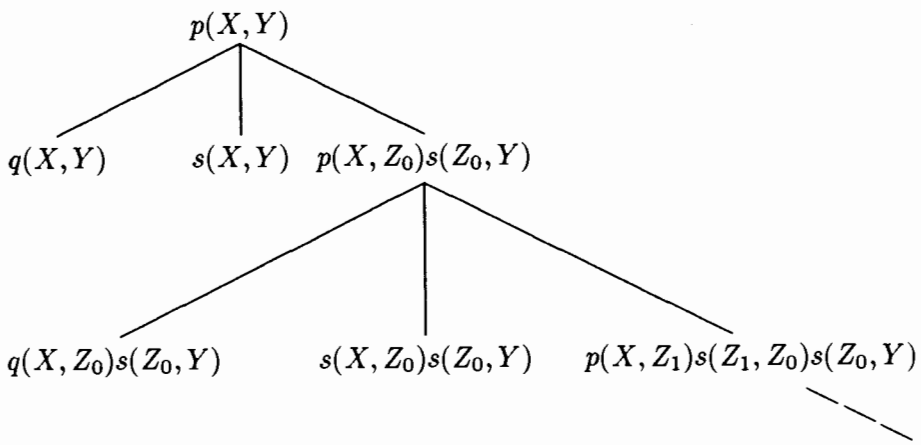


Figure 4.2: A Portion of an Expansion Tree

the tree, we would convert each of the terminal conjunctions into a relational algebra join and take the union of the results of these joins. Thus the value of the relation p would be the union of the relations q, s, the join $(q \bowtie_{q.2=s.1} s)$ and so on. Of course for recursive predicates this is impossible to do directly in practice because the set of terminal nodes is infinite.

The expansion forest of a program tells us how IDB relations can be built up from the extensional database. Therefore, if the extensional database satisfies a set of constraints we can use these to prune the forest.

Example 4.4.3 (cont'd) Suppose the database (made up of the relations s and q) satisifies the constraint $(q(A,B), s(B,C) \rightarrow)$. This constraint says that it is never the case that there is a tuple (a,b) in q and at the same time there is a tuple (b,c) in s. But this is one of the possible ways that a tuple in the relation p can be derived. Therefore, no tuples can be formed from the terminal node $q(X,Z_0)s(Z_0,Y)$ and so the node can be pruned from the expansion tree and removed from the expansion set. □

In general terms, if we have a constraint of the form $(A_1,\ldots,A_n \rightarrow)$ and the residue of it and a node in the expansion tree is empty (i.e. $A_1,\ldots,A_n$ subsumes the node), then the subtree rooted at that node can be deleted from the tree. If the residue is not empty, the situation is more complex but should be investigated as this is what will happen in the majority of cases. In these cases it is clear that one should not delete the subtree rooted at the node. Instead one must apply the residue in some way to each node in this subtree.

Example 4.4.3 (cont'd) Suppose we have the constraint

$$s(A, B), s(B, C) \rightarrow$$

The residues of this and each of the right two nodes shown at the bottom of Figure 4.2 are null, so each of these nodes can be removed. This will lead to the subtree rooted at the rightmost node being deleted. This in turn leads to the tree becoming finite. We have thus obtained a recursive program which is bounded. It is clear that this could not have been achieved by the single rule optimiser. □

So to optimise recursive programs using semantic knowledge we have to do the following (theoretically at least):

1. Convert the program to one or more expansion trees—the expansion forest of the program.
2. Calculate the residues of each node in each tree and the integrity constraints.
3. Prune or alter the tree depending on what residues are found.
4. Convert the forest back into program form.

For non-recursive programs this is possible to implement as it stands. With recursive programs the trees will be infinite. We therefore need a method of calculating and applying the residues to the tree, which does not attempt to generate the tree first. The algorithm will have to determine if an arbitrary conjunction of predicates is a subconjunction of any node in the expansion tree of a recursive predicate. This is the route that further work should take. Once this has been done, and the other components devised, we have to show that any program corresponding to a pruned forest will be semantically equivalent to the program corresponding to the original forest.

4.5 Semantic Optimisation of DatalogIC

We are now in a position to draw together all the various ideas on semantic optimisation and semantic knowledge presented previously and to synthesise a semantic optimisation system for DatalogIC. In Chapter 3, I described various semantic optimisation systems. In Section 4.3, I outlined how we might manage the semantic knowledge for use by an optimiser, and in Section 4.4, I gave some ideas on the optimisation of recursive programs.

In this section, I will give a top-down view of the semantic optimisation system outlining how the various components fit together. This is the global view. I then consider the most important component, the optimisation of a single rule, which I call the local view. This is the only part that has been implemented fully. Figure 4.3 gives an overview of the semantic optimisation system.

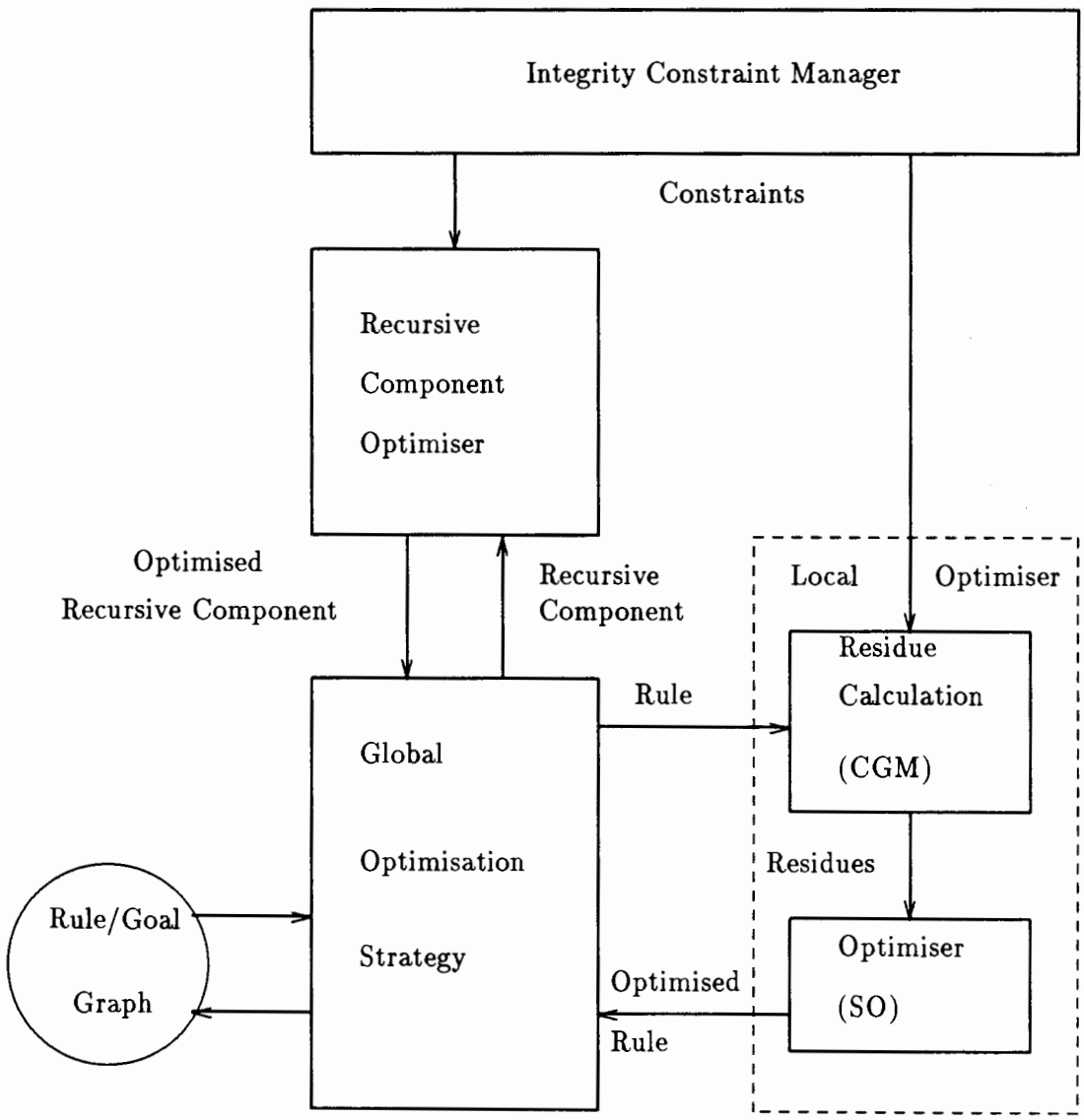


Figure 4.3: Overview of Semantic Optimisation System

4.5.1 Global Optimisation Strategies

Since the global optimiser is concerned with the optimisation of the program as a whole, it interacts with the constraint manager, and controls the optimisation of individual rules

and recursive components. In the DatalogIC system only the simplest global optimiser has been implemented, namely, iteration through the rules of the program, applying the local optimiser to each one.

To design the global optimiser, we need to decide how the components fit together and what strategy the global optimiser will use. We have as our building blocks the following: a method for the optimisation of an individual rule, a scheme for the management of semantic knowledge and a system for the optimisation of recursive components. The task is to link all these together into a single, efficient system. I shall start with just the rule optimiser and show how we might control this, after which I shall incorporate the constraint manager and recursive component optimiser. It is important to note that the building blocks are not independent since they affect each other.

As the rule is the basic construct of a Datalog program, it is natural that the division between local and global optimisation is made here. With recursive programs we can extend this and say that the global optimiser does not see the optimisation of a single rule as atomic, but sees the optimisation of a single *component* (which in recursive programs could contain more than one rule) as being the atomic operation.

As well as controlling the optimisation process, the global optimiser has to propagate the changes resulting from a rule being labeled as redundant by the local optimiser. To work out what needs to be done we have to understand what the implications of a rule being made redundant are. If a rule is made redundant it means that no tuples are going to be generated by this rule, and it can be deleted from the rule list of the definition containing the rule. This in turn could lead to the rule list for a definition being made empty. An empty rule list means that the relation defined by the definition will always be empty. An empty definition does *not* mean that any occurrences of the predicate itself are redundant, in other words always true. Instead, it means that any predicate occurrence will always be false, so rules containing an occurrence are redundant. It is clear that we have to choose a global optimisation strategy that keeps the time spent carrying out this propagation to a minimum.

As with evaluation systems, we can have bottom-up optimisation strategies and top-down optimisation strategies (amongst others). The bottom-up scheme is a post-order traversal of the definitions in the condensed rule/goal graph (which, as mentioned in Section 2.2.5, is actually a tree). We first optimise definitions that depend only on the

EDB predicates, then move up the tree to optimise the next level of definitions, and so on. The top-down scheme starts from the query and moves down the rule/goal graph until the EDB predicates are reached. A problem with the top-down method is that if we optimise a rule r after the rules which depend on it (higher up the rule/goal graph) then, if r is made redundant, we will have to optimise the higher rules again. The solution is to optimise the program by working in the same direction as the redundancy changes are propagated. This is what the bottom-up strategy does.

We now consider the incorporation of semantic knowledge inference. Although we aim to have the inference of new knowledge separate from the optimiser, the global optimiser must take into account the inference process. If this is not done, then the constraint manager may infer a piece of semantic knowledge that could be used to optimise a rule that has already been considered by the optimiser. If the inference process involves only the upward inference of semantic knowledge, then a bottom-up optimisation strategy is the best. However, if we wish to incorporate a mechanism which pushes conditions down the rule/goal graph from the query (this is the same as pushing selects inwards; and is discussed in Section 4.3.2), then we are faced with the same problem as above. The optimisation of a rule could give us a condition which we can push down the rule/goal graph, and in turn may enable us to optimise further the rules which we have already passed, as we optimised up the rule/goal graph. An initial solution to this is to use the blackboard technique from Artificial Intelligence, and devise a set of production-rule heuristics.

The easiest way to incorporate the recursive rule optimiser is to limit it to the optimisation of a single recursive component at a time. Once a recursive optimisation scheme has been formalised, a study must be made to determine whether or not the application of the optimiser to one component at a time gives us an optimised program that is equivalent to a program obtained by optimising the whole program at once.

4.5.2 Local Optimisation

I now discuss the design of the local optimisation module, drawing on the concepts detailed in Chapter 3. The task of the local optimiser can be stated as follows: given a rule and the semantic knowledge provided with the program, plus any knowledge that has been infered, the optimiser must optimise the rule into a simpler, but semantically equivalent,

rule. It is the task of the global optimiser to find which rule the local optimiser must work on, as has been discussed above. I first outline why we cannot use the CGM and the SO systems, given in Sections 3.2 and 3.3 and as presented in [CGM87, SO89], individually to construct the rule optimiser. Then I show that we can, however, use the two together to build the local optimiser, as each resolves omissions present in the other. Finally, I outline the actual algorithm which has been implemented fully in the DatalogIC system, and give an example of what it does.

The CGM semantic optimisation system takes as input a set of Horn clause integrity constraints, a query and an intensional database of rules. The Horn clauses are merged with the rules to form residue sets, one set for each rule. When the query is compiled, into a set of EDB conjunctions, the residue sets of the rules used in the compilation are combined to form a residue set, one for each of the conjunctions. The residues are then applied to the compiled query to yield one or more semantically equivalent, but simpler, queries.

The problem with the CGM system as described is that it relies on an evaluation system which compiles queries down to conjunctions of EDB predicates. Since we are designing a system that will handle recursive rules, it is not always possible to compile queries in this way. Instead, we want the system to optimise rules individually. Apart from this, it is always a good idea to have the optimisation system independent of a particular type of evaluation system. Also, in the presentation of this system there is a lack of detail on how the residues can be applied, especially when compared to the details given on residue calculation.

The SO system was originally designed from a relational database viewpoint. It takes as input a set of value and subset constraints, and a relational calculus query. This query is then converted into a graph form. The constraints are used to convert the query graph into a semantically equivalent, but simpler, graph. The new graph is then converted back to relational form. Thus this system seems ideal for applying constraints to the rules. However, the problem with this system is the language mismatch which can be divided into three aspects: The system works with relational calculus queries only and not Horn clause form queries. There is no concept of rules which define predicates in terms of other predicates. Finally, the constraints are not in Horn clause form. In Section 3.3, I indicated the form that SO does accept, but in the description of the system Horn clause queries

and constraints are used and I will show how the translation can be made shortly.

It is clear therefore that both SO and CGM have limitations. However we can resolve these problems as I will now show. First the language mismatch problem. Recall that in Section 3.1.3 I described how to translate relational constraints to Horn clauses. We now have to translate in the other direction from Horn clauses into the form accepted by the SO system. This translation is necessary because of the different ways that arguments are treated: in Horn clauses it is the position of an argument in the predicate that is important, while in relational-type constraints it is the name that is important. So to convert a Horn clause to relational form we first have to fully rectify the clause, and then map the variables in the clause onto the attributes of the relations involved. Since we need the constraints to optimise a particular rule, it is only necessary to map the variables in the constraint onto the variables used in the fully rectified rule, and not onto the attributes of the underlying relations.

Example 4.5.1 Suppose we have the Horn clause constraint

$$employee(X, Y, Z), Y = admin \rightarrow Z > 15.$$

If the *employee* relation has attributes $Name$, $Class$ and Sal then we can translate the constraint into

$$employee.Class = admin \rightarrow employee.Sal > 15$$

This is the type of constraint that the SO system will accept. Note that there are only evaluable predicates in the constraint. If we were going to use the constraint to optimise the rule

$$high\_salaries(X, Y) \ :- \ dept(X, Y), employee(X, admin, Z), Z > 10.$$

then we do not need to know the names of the attributes of the relations. Instead we rectify the rule to

$$\begin{aligned} high\_salaries(X_0, Y_0) \ :- \ & dept(X_1, Y_1), employee(X_2, C_2, Z_2), X_0 = X_1, X_1 = X_2, \\ & C_2 = admin, Z_2 > 10. \end{aligned}$$

and translate the constraint into

$$C_2 = admin \rightarrow Z_2 > 15$$

using the variables in the *employee* predicate. We have done exactly the same as in the first translation, except that C_2 serves as an alias for *Class* and Z_2 serves as an alias for *Sal*. If we do not rectify the rule and carry out this variable mapping then ambiguities can arise. □

The translation process in the above example is actually residue calculation. To translate a Horn clause constraint into a form usable by the SO system, we calculate the residue of it and the rule we are going to optimise. If any residue contains an occurrence of a non-evaluable predicate then it can not be used by the SO system, and so it is deleted from the residue set. Therefore, the residue calculation part of CGM is ideal to help us to link the SO system into the DatalogIC environment.

Since queries are syntactically the same as rule bodies, applying the SO system to rule bodies can be done with no changes being necessary. However, the problem with optimising the rules directly, instead of the compiled queries, is that some of the residue forms will not be used and will be wasted, as I mentioned in Section 3.2.3. However, the constraint inference process, which I presented in Section 4.3, will go some way toward solving this. The following example illustrates the problem and its solution.

Example 4.5.2 Suppose we have the following rules

$$\begin{aligned} p(X,Y) &:- q(X,Z), r(Z,Y). \\ s(X,Y) &:- t(X,Y). \end{aligned}$$

with query

$$?- p(X,Y), s(X,Y).$$

and constraint

$$q(A,B), t(A,C) \rightarrow$$

In a sense the constraint links together two branches of the rule/goal tree. The CGM system would compile the query down to

$$(q(X,Z), r(Z,Y), t(X,Y))\ \{t(X,C) \rightarrow, q(X,B) \rightarrow\}$$

where the two residues are shown in braces. Using either of the two residues we would find that the query is FALSE. The DatalogIC system as implemented is not able to do this, since the queries are not compiled, and the optimiser, which considers each rule individually,

will not be able to rewrite either of the rules. The solution is to propagate the residues up to the query in some way. This can be achieved by infering the following constraints using the rule, R_4, given in Section 4.3.2

$$p(X,Y), t(X,C) \rightarrow$$
$$p(X,Y), s(X,C) \rightarrow$$

The second constraint can be used to optimise the query directly with no compilation being necessary. □

One can see that the SO and CGM systems dovetail nicely into each other: the residue calculation algorithm of CGM can convert constraints into the form usable by SO; the SO system can serve as the residue application part of the CGM system. The complete local optimisation process is given in Figure 4.4.

Algorithm: Local Optimisation of a Rule.
INPUT: Rule and Integrity Constraints.
OUTPUT: Optimised Rule.

BEGIN
  For each constraint, I
    Calculate residues of constraint I and rule. (Algorithm 3.2)
  Delete those residues which contain at least one non-evaluable predicate.
  Apply residues to rule (Algorithm A.15) as follows
    Form query graph from the rule body.
    Form condensed canonical form.
    Apply residues to form semantic expansion graph.
    Remove redundant relations and edges.
    Convert back to rule form.
END

Figure 4.4: Local Optimisation

Example 4.5.3 Here is a sample optimisation session. When a program is processed by the DatalogIC system a script of what has occurred is written. Below is an annotated example of one such script, using the program given in Example 4.1.1

```
                Listing for Datalog-IC program t3
-----------------------------------------

1:    EXT emp(Name,Edept,Sal){
```

```
   2:            // All employees earn more than 5000.
   3:            IC -> Sal > 5000.
   4:            // A manager of a department earns more than the members of
   5:            // the department.
   6:            IC manager(Mname,Edept,S) -> S > Sal .
   7:    }
   8:
   9:    EXT manager(Mname,Dept,Sal){
  10:    }
  11:
  12:    // This gives us managers earning less than 4000, and their
employees.
  13:    INT lowmanager(X,Y){
  14:           :-  manager(X,Dept,S), emp(Y,Dept,S2), S < 4000.
  15:    }
  16:
  17:
  18:    ?- lowmanager(!Mnane,?Ename).

Program has been parsed (0 errors 0 warnings)

Optimising
----------

Calculating residues for rule:
lowmanager (X,Y)  :-  manager (  X, Dept, S ),  emp (  Y, Dept, S2 ),
                      S < 4000,
Using integrity constraint:
IC:  emp (  Name, Edept, Sal ) manager (  Mname, Edept, S ) S <=  Sal,  ->
```

Note how the head of the definition has been added to the body of the constraint on line 6, and the head of the constraint has been negated and moved into the body.

```
Residues:
IC:  S <=  S2,  ->

IC:  S2 >=  S,  ->



Calculating residues for rule:
lowmanager (X,Y)  :-  manager (  X, Dept, S ),  emp (  Y, Dept, S2 ),
                      S < 4000,
Using integrity constraint:
IC:  emp (  Name, Edept, Sal ) Sal <= 5000,  ->

Residues:
IC:  S2 <= 5000,  ->
```

The system now forms the semantic expansion and finds that the new edges cause the body of the rule to be false and thus the rule to be redundant.

```
Using IC on line 2. adding edge emp.S2  >  5000
Using IC on line 5. adding edge emp.S2  <  manager.S

Optimisation finished on rule
(Rule redundant)
lowmanager (X,Y)  :-  manager (  X, Dept, S ),  emp (  Y, Dept, S2 ),
                      S < 4000,
End of Optimisation
--------------------
```

□

4.6 Conclusion

In this Chapter, I have introduced the language DatalogIC, which is the same as Datalog except that semantic knowledge about the predicates can be expressed. I have proposed a full semantic optimisation system for it. This system is composed of the global optimiser, the local optimiser, which optimises a single rule, the recursive component optimiser and the integrity constraint manager. The key part of this chapter has been the discussion on the local optimiser, Section 4.5.2, which has been implemented fully. In the next Chapter I discuss the rest of the implementation.

Chapter 5

Implementation of DatalogIC

This chapter outlines the implementation of the DatalogIC system introduced in Chapter 4. The system has been implemented in the C programming language, on the UNIX operating system and with the Oracle relational database system as the backend. As well as implementing some of the ideas presented in previous chapters, I have attempted, as a secondary aim, to provide the foundations for further work on Datalog programs within the department (which has already been started). The approach taken with this chapter is to give more of a flavour of the system, and to show how all the ideas presented in the previous chapters fit together, rather than to list the data structures and the algorithms used. I leave for Appendix A the details of the data structures and how programs are held in the system. Figure 5.1 shows the overall structure of the system. In the next section, I describe each part of the system shown in the diagram, except those parts that warrant a section for themselves. In Section 5.2, I discuss the user interface module and the commands available to the user. Section 5.3 covers how Datalog programs are evaluated, dealing with the Semi-Naive evaluation algorithm and the conversion of rules into SQL. The final section, Section 5.4, gives the results of tests I carried out on the system.

5.1 System Overview

- **Rule/Goal Graph** The rule/goal graph is the central data structure of the system. The functions in this module manipulate the graph prior to query processing, and check that certain syntactic conditions, such as safety, hold. The main operations executed out by the rule/goal graph module during the parsing process are the

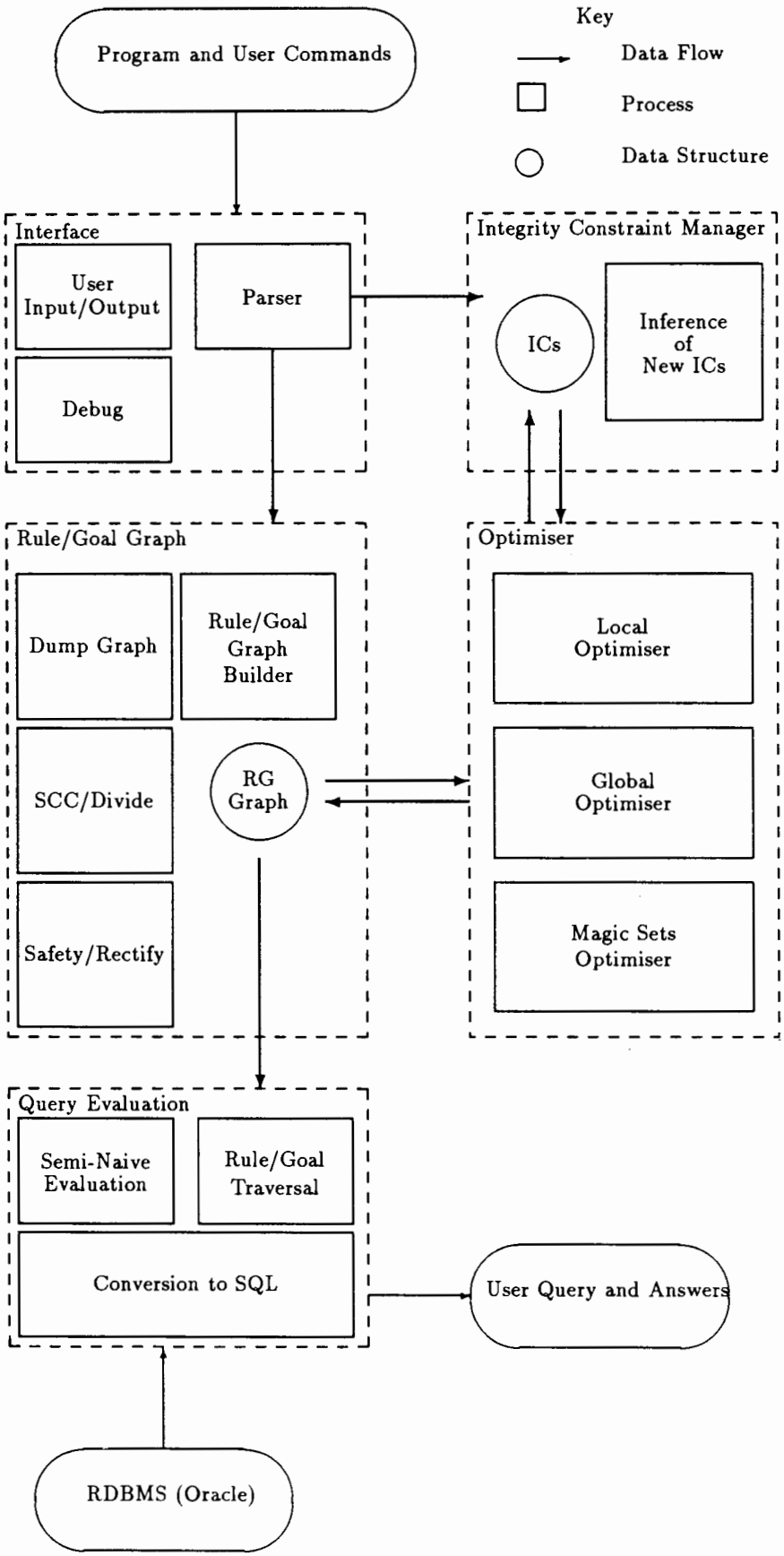


Figure 5.1: Overview of the DatalogIC System

rectification and safety checking algorithms (labeled Safety/Rectify in the figure), as well as the algorithms to form the strongly connected components and divide the rules into non-recursive and recursive parts (labeled SCC/Divide). All these are outlined in Appendix A and have been motivated in Chapter 2. Section 5.2 shows what operations are carried out on the program during the parsing process, and Section A.1 shows how a program is held, in rule/goal graph form, in the system.

- **Integrity Constraint Manager** This organises integrity constraints so that they are used efficiently by the optimiser, as has been discussed in Section 4.3. It processes requests for constraints from the optimiser, passing the latter the minimum number of constraints required. When a request is made by the single rule optimiser, the constraint manager will build a list of constraints that have predicates that occur in both the rule and the constraint. This list is ordered on the number of predicates in the constraint that do not occur in the rule, and is returned to the optimiser. Appendix A.1.3 describes how constraints are held in the manager.

- **Optimiser** This takes semantic knowledge from the constraint manager, and the program from the the rule/goal module, and carries out semantic optimisation on the program. The semantic optimisation system is divided into the global optimiser and the local optimiser. As mentioned in Section 4.5, only the simplest global optimiser has been implemented. The global optimiser merely serves as the driver for the single rule optimiser, which has been implemented fully and is a fusion of the SO and CGM systems presented in Chapter 3. The standard Magic Sets optimisation has also been implemented, the algorithm for which is given in Section A.3.3. Using the Magic Sets system one can investigate the interaction between this syntactic optimisation system and semantic optimisation. This is another direction that further work could take. Examples of what both optimisers can do are given later on in Section 5.4.

- **Miscellaneous** Since both the optimiser and rule/goal modules use graph-theoretic algorithms, a reusable graph module was created. This finds the strongly connected components of a graph and forms the transitive reduction of a graph. A memory manager was created to monitor how much memory is being used and trap any memory allocation errors.

5.2 User Interface

This handles all the interaction between the programmer and the system, and is composed of three parts: the parser, the debugger and the user input/output routines.

The parser, which parses DatalogIC programs, as well as user commands, is written using the UNIX tools, Lex and YACC. The commands available to the user are as follows:

- **`parse [program name]`** Parses the given DatalogIC program and reports errors. The program then becomes the current program.
- **`compile`** Compiles the current program into SQL statements (see Section 5.3).
- **`optimise`** Optimises the current program (as described in Section 4.5).
- **`magic`** Carries out the Magic Sets optimisation on the current program.
- **`query [query name]`** Executes the named query.
- **`set [option name] [value]`** Sets the named option to the given value. Options include setting the level of debugging, where debugging information is to go, whether or not the system should connect to Oracle and the destination of all user output.
- **`quit`** Exit the DatalogIC system.

When the **`parse`** command is entered the input to the parser is switched to the named program file which is then read in. The parsing process is as follows:

- Initialise counters and set up files.
- Parse the program. As each Horn clause (rule or constraint) is read in it is rectified.
- Report errors; if there are none, continue.
- For each definition, build a list of all the definitions which depend on it (the child list) .
- Find the recursive components using the child list.
- Divide the rule list into a list of recursive and non-recursive rules.
- Check rule safety and inform the user of any unsafe rules.

On encountering a particular non-terminal in the grammar (such as a definition or a rule), an action is carried out. Most of the actions pass the structure just parsed to the rule/goal graph, or to the integrity constraint manager. Appendix A.2 gives some more details on the parsing process. A log file, showing a summary of the various actions carried out on the current program, is created (see Section 4.5 for an example of such a file).

Since the system is in its infancy and liable to be built upon, extensive debugging facilities were provided. A universal debugging function takes an arbitrarily long parameter list giving instructions on the structures to print. This function then calls the necessary functions in the Dump Graph part of the rule/goal graph module to print the required structures.

The aim of the user input/output submodule is to handle all information that passes to and from the user. Having this single interface point will make porting to other systems, such as the X Windows System [1] or microcomputer-based environment, far easier.

5.3 Query Evaluation

The principal part of query evaluation is the compilation of the rule/goal graph into SQL statements. In Section 2.3.1, I discussed the merits and demerits of compilation as an evaluation technique. Compilation in DatalogIC is done by a post-order traversal [2] of the reduced dependency tree, starting with the query. The reduced definition tree has three types of nodes: the nodes for the EDB definitions, the nodes for non-recursive definitions and the nodes for the recursive components (which contain more than one definition). Each type of node is treated differently.

For both the non-recursive and recursive cases we convert rules into SQL expressions; I shall call the function which does this $SQL(\ )$. For the fully rectified rule

$$p(\overline{X}) \ :- \ q_1(\overline{X}_1), \ldots, q_n(\overline{X}_n), \delta$$

where δ is a list of comparisons involving the only the variables mentioned in the non-evaluable predicates [3] we obtain the SQL expression

[1] X Windows is a trademark of Massachusetts Institute of Technology

[2] We evaluate a node once we have dealt with all its children nodes.

[3] Note that this is an extra restriction on the syntax of permissible rules. Further work will have to be done so that the system accepts rules like $p(X) : -r(X,Y), X = Z, Z = Y$.

select $alias_{m_1}.y_{m_1}, \ldots, alias_{m_k}.y_{m_k}$ from q_1 $alias_1, \ldots, q_n$ $alias_n$ where δ'

where:

- Each variable in a predicate occurrence is mapped onto its *underlying attribute* taken from the predicate's definition. This is the attribute that appears in the same position in the head of the definition as the variable does in the predicate occurrence (see the end of Example 5.3.1 for an example of this). As the rule is fully-rectified each variable in the rule has a unique underlying attribute.

- $alias_i$ is a unique alias for q_i. We have to use aliases as there may be more than one occurrence of a predicate in the body of the rule.

- y_{m_i} is the underlying attribute of the first body variable that is equated to the head variable x_{m_i} while $alias_{m_i}$ is the alias of the predicate occurrence (i.e. q_{m_i}) in which the body variable appears.

- δ' is the same as δ except that every variable, x_j is replaced with $alias_i.x_j$, where $alias_i$ is the alias of the predicate occurrence containing x_j, and the comparisons in δ' are linked together with "and"s

On encountering a non-recursive intensional definition, the system generates a SQL **create view** statement. For the definition D with attributes $x_1, \ldots, x_n$, and with rules $R_1, \ldots R_n$, $SQL(D)$ is the expression:

create view D (char(20) x_1,..., char(20) x_n) as
$SQL(R_1)$ union ...union $SQL(R_n)$.

Only one underlying database type has been used, that of strings of length 20, so numeric terms are filtered when they are used (see Example 5.3.1).

For an extensional node in the tree we do nothing, as it is assumed that the corresponding relation exists in the database, with the same attribute names as those appearing in the definition head.

Example 5.3.1 The non-recursive program

```
INT job(Name,Dept) {
        :- emp(Name,Eno), Eno < 10, manager(Name,Dept).
```

```
        :- emp(Name,Eno), dept(Eno,Dept).
}

EXT manager(Name,Dept){}
EXT emp(Ename,Enumber){}
EXT dept(Enum,Dept){}

?- job(?Name, ?Dept).
```

is converted into the SQL expressions:

```
CREATE VIEW job (Name, Dept) AS
   SELECT emp1.Ename,  manager3.Dept
     FROM emp emp1, manager manager3
     WHERE emp1.Ename  =  manager3.Name
       AND to_number( emp1.Enumber ) < 10
   UNION
   SELECT emp1.Ename, dept2.Dept
     FROM emp emp1, dept dept2
     WHERE emp1.Enumber =  dept2.Enum;

CREATE VIEW Query0 ( Name ,  Dept ) AS
  SELECT job1.Name, job1.Dept FROM job job1;
```

In the first rule the underlying attribute for the first occurrence of $Name$ is $Ename$, while for the first occurrence of Eno it is $Enumber$. These can be determined by looking at the definition for emp. □

On encountering a recursive node in the reduced dependency tree, the system will run Semi-Naive evaluation on the definitions in the component. This implies that the DatalogIC compilation is not true compilation as evaluation is carried out and the system has to access the extensional database. However, this can be overcome as I will mention at the end of this section.

The first stage is to set up the temporary tables and views required. Given a recursive component containing the definitions $D_1, \ldots D_n$ we set up the following temporary tables

and views for each definition:

- $D_i.delta$—the difference between the current value of the relation and the value of the relation from the previous iteration.
- $D_i.relation$—the current value of the relation.
- $D_i.baseview$—the base view used to calculate the initial value for D_i.
- $D_i.joinview$—the join view used within the `while` loop, and is equivalent to the $eval\_incr()$ function given in Section 2.3. This will reference $D_i.delta$ when values for the D_i relation are required.

The base view is built using the list of non-recursive rules, in the same manner as for a non-recursive definition. The join view is built using the list of recursive rules in a similar way, except that we stop at any predicate that is in the current component in order to prevent cycling. Note that the building of these views could lead to a suspension of work on the recursive component while work is done on the parent definitions of the component (the definitions that the component depends on).

Example 5.3.2 For the *path* example the base view is the following expression:

```
create view pathbaseview (char(20) X, char(20) Y)
  select link1.X,link1.Y from link link1.
```

while the join view looks like:

```
create view pathjoinview (char(20) X, char(20) Y)
  select pathdelta1.X, link2.Y from pathdelta pathdelta1, link link2
  where pathdelta1.Y = link2.X.
```

□

The Semi-Naive algorithm is shown in Figure 5.2. The function $Eval\_SQL()$ evaluates an SQL expression.

As noted in Section 2.3 the algorithm only works properly with linear recursive rules. The above process is not true compilation, as we evaluate the recursive components before we proceed with the rest of the compilation. However, this can be overcome if the actual evaluation part of the above algorithm, the `while` loop, is not executed until query time.

Algorithm: Semi-Naive Evaluation

INPUT:
List of definitions in the recursive component and their rule sets.
Values for non-recursive relations involved.
OUTPUT:
Values for each of the relations in the component.

BEGIN
Set up the temporary tables and views as given above.
For every definition D_i
$D_i.delta = Eval(D_i.baseview)$
$D_i.relation = D_i.delta.$
endfor
While at least one $D_j.delta$ is non-empty
For every defn D_i
$D_i.temp = Eval\_SQL(D_i.joinview)$
endfor
For every defn D_i
/* This removes the duplicate tuples */
$D_i.delta = D_i.temp - D_i.relation$
$D_i.relation = D_i.relation \cup D_i.delta$
endfor
endwhile
END

Figure 5.2: Semi-Naive Evaluation

To do this, we keep a record of all the information required by the `while` loop, such as names of definitions in the components, which will be used at query evaluation time. This information could be stored in the database itself. The efficiency of the above system also needs some investigating; for instance, storage of partial results should be considered. The next section gives some operating times for Semi-Naive evaluation.

5.4 Empirical Results

In this section, I present a few tests I made on the system in order to confirm that semantic optimisation does give some improvement in query processing time. More thorough testing is required to determine the exact improvement that can be gained through semantic optimisation. First, I discuss the test database definition language I used to build test databases, then I give test results for non-recursive and recursive programs, and finally I discuss the results. For each test, I point out how the optimisation leads to the results obtained. More general observations are left to Section 5.4.4

5.4.1 Test Database Definition Language

In order to get a good spread of query evaluation times, I had to generate fairly large test databases and this is impractical to do by hand. I therefore augmented the DatalogIC language with a test database definition language (TDDL). Two types of definition are possible: one for non-recursive test databases and another for recursive test databases.

Example 5.4.1 The following test database definition was used to generate the database for the join elimination test:

```
EXT deptman(Dept,Mname){
        #300 unique unique  joinwith 10:10 deptemp
}

EXT deptemp(Ename,Dept){
        #3000 unique unique
}
```

For the *deptman* relation the system will generate 300 tuples, both columns of which will have unique values. The `joinwith a:b` part means that, for every tuple generated for

the first relation (i.e. *deptman*), between a and b tuples will be generated for the second relation (i.e. *deptemp*). In this example exactly 10 tuples will be generated. These tuples will have the same value in the *Dept* column position as the *deptman* tuple (since the columns have the same name) and a unique value for the *Ename* column. Columns can also be specified to have random values from a particular range, or constant values. □

To generate databases for the recursive tests, I used the relation specification ideas from [BR86]. There an analytical study is made of the evaluation algorithms discussed by investigating what would occur if they were run on various types of relations which are defined by four parameters. The relations are binary relations with a tuple (x, y) viewed as an arc between two nodes, x and y. The four defining parameters are

- **Fan out**: The number of arcs leading from a node (the number of children a node has).
- **Duplication**: The number of arcs coming into a node (the number of parents a node has).
- **Height**: The length of the longest path in the relation.
- **Base length**: The number of nodes with no arcs coming into them.

These parameters can be used to specify relations which have a tree-like or a cylindrical-like structure. A tree with height h and n children for each node will be specified by the parameter list $(n, 1, h, 1)$. The cylindrical relation shown in Figure 5.3 will have a parameter list of $(2, 2, 4, 5)$. Each tuple is a pair of elements of the form $A(l, w)$ where l is the level of the node and w its position in the level. For example, the root of a tree will always be $A(0, 0)$ and the link between the root and its second child will be specified by the tuple $(A(0, 0), A(1, 1))$ in the relation.

5.4.2 Non-Recursive Programs

Four tests were carried out on various non-recursive programs. Similar tests have been reported in [SO89], and have been given in [Kin81] as examples of what semantic optimisation should do (see Section 3.1.4). The tests are index introduction, join elimination, restriction elimination and scan reduction.

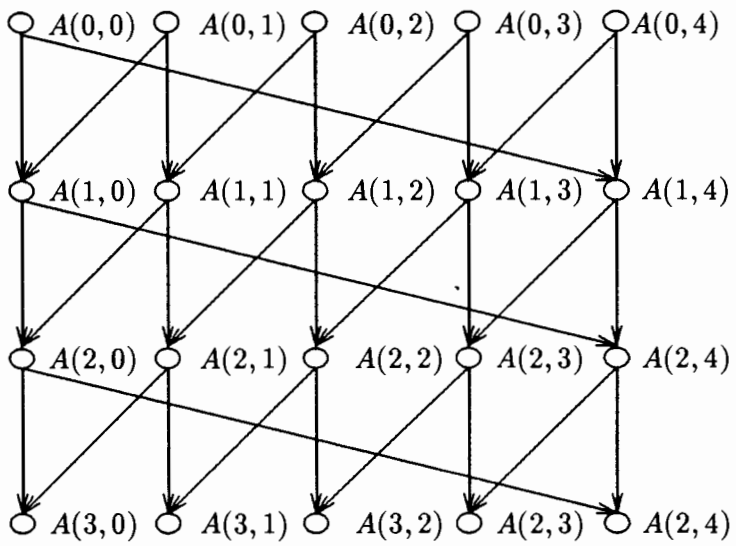


Figure 5.3: A Cylindrical Database

For each test, I give the program used and a table showing the time spent optimising the program, compiling the program and evaluating it. If the entry for optimisation is 0, then the program was not optimised. The times for each column in a table are the averages obtained from 10 test runs. It should be pointed out that the times spent on compilation vary considerably in some cases without apparent reason. I suspect that this has something to do with the manner in which Oracle creates and drops tables, which is slow compared with evaluation. Thus I do not consider these anomalies to be significant.

The test programs used the following extensional predicate definitions:

```
EXT employee(Name,Class,Sal){
        // All, and only, Managers earn more than 10 thousand
        IC Class = 'manager'  -> Sal >= 10000.
        IC Sal >= 10000 -> Class = 'manager'.
}

// This gives the members of a department
EXT deptemp(Name,Dept) {
        // An employee can only be in one department
        IC deptemp(EName,D2) -> Dept=D2
}
```

```
// This gives the managers of a department
EXT deptman(Dept,Name){
        // All managers manage someone.
        IC -> deptemp(_,Dept).
        // Only managers manage
        IC employee(Name,Class,X3) -> Class = 'manager' .
}
```

Index Introduction

This test investigates what happens when a restriction on an indexed attribute is introduced into a rule by the optimiser. The following intensional database and query were used.

```
INT high(Name,Class,Sal){
        :- employee(Name,Class,Sal), Sal > 15000.
}
```

```
?- high(?Name,?Class,?Sal).
```

The TDDL was used to generate three different *employee* relations, with 5%, 10% and 25% of the tuples specified as having the *Class* column set to "*manager*". An index was created for the relation on the *Class* column, and the total number of tuples in the relation was 10 000. Since the constraints tell us that us that anyone who has a salary of over 15 000 is a manager, the condition $Class = manager$ is added to the rule body by the system.

The results for this test are shown in Table 5.1. Since the *Class* attribute is indexed, selecting out the manager tuples first, and then testing the value of *Sal* is quicker than testing the value of *Sal* on all the tuples. However, this only occurs when the manager tuples form a small percentage of the whole database. At 5% we see that the query evaluation time for the optimised program is just over half the evaluation time for the unoptimised program, while at 10% the evaluation time for the optimised program is 67% of the time for an unoptimised program. At percentages larger that 25% the index restriction starts to become unprofitable.

Activity	Duration (secs)					
	5%		10%		25%	
Optimisation	0.0	2.7	0.0	2.3	0.0	2.6
Compilation	14.7	16.0	13.9	15.4	14.2	15.0
Query Evaluation	14	7.3	15.8	10.6	22.2	22.3
Total	28.7	26.0	29.7	28.3	36.4	39.0

Table 5.1: Times for Index Introduction Test on Three Employee Relations

Join Elimination

For this test we investigate what happens when predicates, and thus joins, are eliminated from the body of a rule. The following intensional database and query were used:

```
// Man1 and Man2 manage the same employee
INT managersame(Man1,Man2){
        :- deptman(D1,Man1), deptman(D2,Man2),
                deptemp(Emp,D1), deptemp(Emp,D2).
}

?- managesame(?X,?Y).
```

The extensional database used is the one described in Example 5.4.1 Optimising this rule will lead to the two occurrences of the *deptemp* relation being dropped. First, $D1$ and $D2$ are equated by the constraint which tells us that an employee only works for one department. This leads to one of the duplicate occurrences of *deptemp* being dropped. The second occurrence of *deptemp* is now dangling and is dropped (by virtue of the constraint $depman(Dept, \_) \rightarrow deptemp(\_, Dept)$). The optimised rule looks like

$$managersame(Man1, Man2) : -deptman(D1, Man1), deptman(D1, Man2).$$

In other words, the query will simply return all managers managing the same department. From the test database definition, Example 5.4.1, one can see that an employee has only one manager. Hence only tuples of the form $< man, man >$ will be returned, where *man* is the name of a manager. For the optimised query 300 tuples of this form are returned, while for the unoptimised query 3000 tuples are returned. The difference arises because the latter result contains duplicate rows.

Activity	Duration (secs)	
Optimisation	0.0	28.9
Compilation	12.9	14.5
Query Evaluation	81.6	5.7
Total	94.5	49.1

Table 5.2: Times for Join Elimination Test

The times for this optimisation, given in Table 5.2, show that the elimination of two joins reduces query time by approximately 95%. Even if one takes into account the fact that the unoptimised query returns duplicates, the saving is still considerable. This gain is expected since the join is the most costly database operation.

Restriction Elimination

In this test, we see what happens when a restriction is eliminated from a rule body. The following intensional database and query were used:

```
INT lowsal(Man){
        :- employee(Man,Class,Sal), Class = 'manager', Sal > 5000.
}

?- lowsal(?Man).
```

The *employee* relation contained 1000 tuples with the *Class* column set to "manager" and 10 000 tuples with the *Class* column set to "clerk". Optimisation of the rule results in the restriction $Sal > 5000$ being eliminated, as a constraint specifies that all managers earn over 10 000.

Activity	Duration (secs)	
Optimisation	0.0	1.8
Compilation	14.0	14.0
Query Evaluation	17.6	16.9
Total	31.6	32.7

Table 5.3: Times for Restriction Elimination Test

The results for this test are shown in Table 5.3. The saving gained in query time by eliminating the unprofitable restrictions is only about 4%. This is as one would expect because a comparison test is not expensive in relation to the other evaluation operations.

Scan Reduction

This test investigates what happens when we introduce a restriction on a relation involved in a join. The intensional database and query are:

```
INT manager(Name,Dept,Sal,Class){
        :-  employee(Name,Class,Sal),deptman(Dept,Name).
}

?-manager(?Name,?Dept,?Sal,?Class).
```

Using the constraint, which tells us that only managers appear in *deptman*, the program is optimised to:

```
INT manager(Name,Dept,Sal,Class){
        :-  employee(Name,Class,Sal),deptman(Dept,Name), Class = 'manager'.
}
```

As with the index introduction test I looked at databases with 5%, 10%, and 25% of the tuples in the *employee* relation having the *Class* column set to "manager". The added restriction on the *employee* relation will restrict the number of tuples that Oracle will have to handle in the join with the *deptman* relation.

Activity	Duration (secs)					
	5%		10%		25%	
Optimisation	0.0	4.7	0.0	4.3	0.0	3.2
Compilation	15.3	17.9	17.9	17.7	15.1	17.6
Query Evaluation	56.9	22.5	72.6	37.1	129.5	97.5
Total	72.2	45.1	90.5	59.1	144.6	118.3

Table 5.4: Times for Scan Reduction Test on Three Databases

The times for this test are shown in Table 5.4. The saving gained by this optimisation decreases as the percentage of manager tuples in the *employee* relation increases. With an

optimised program the query time is more than halved when the percentage is less than 10%, and at 25% the query time is cut by a quarter.

5.4.3 Recursive Programs

Magic Optimisation

The aim of this test is to see how Magic Sets optimisation reduces the time spent evaluating a simple recursive program. The following program was used:

```
EXT link(X,Y){}

INT path(X,Y){
        :- link(X,Y).
        :- path(X,Z), link(Z,Y).
}

?- path(?X,!Y), Y = 'A(10,3)'.
```

Two versions of the *link* relation were generated. The first was a binary tree of height 10, with specification $(2, 1, 10, 1)$, containing 2046 $(= 2^{11} - 2)$ tuples. The second relation was a cylinder of width 10, height 10 and duplication/fanout of 3 which contained 100 tuples. For the first relation 10 tuples were returned as answers, while for the second 84 tuples were returned.

Activity	Duration of Activities (secs)			
	Tree		Cylinder	
Optimisation	0.0	2.0	0.0	2.0
Semi-Naive	1012.2	203.4	296.2	232.4
Query Evaluation	12.4	1.0	3.8	2.8
Total	1024.6	206.4	300.0	237.2

Table 5.5: Times for Magic Sets Optimisation on Two Test Databases

The times for these tests are shown in Table 5.5. Recall that the Magic Sets optimisation will rewrite the program so that it first generates the set of relevant facts. This set is then used to complete the evaluation. The benefits gained by Magic Sets optimisation for

the tree database are great, as the set of relevant facts for the query is small (10 tuples). For the cylinder database, the payoff is not as great as the set of relevant facts is much larger (84 tuples). Recall that with recursive programs most of the work is done during the compilation phase, since the Semi-Naive algorithm as implemented involves evaluation; hence the large compilation times in Table 5.5 (especially the first column). Query evaluation on the other hand is just the evaluation of a single SQL statement; the query itself.

5.4.4 Conclusion

In all cases, the optimisation of the programs gives some improvement in query processing time. This includes the scan reduction program, where one would think an extra comparison test would not make much difference. The biggest saving came, as one would expect, from the join elimination test and Magic Sets.

A large part of the total times shown in the tables is spent compiling the program– this is mainly because a lot of time is spent by Oracle adding view definitions, dropping tables and creating tables. Fortunately compilation is done only once per query form. Optimisation is also time consuming but this too is done only once per query, so the time saved should outweigh the optimisation time. The final conclusion is that semantic optimisation does improve the performance of queries. Perhaps if the underlying database were tuned properly the savings would be even greater.

Chapter 6

Conclusion and Further Work

6.1 Conclusion

This thesis can be considered as being comprised of two phases: analysis and synthesis. In the analysis phase, Chapters 2 and 3, I first introduced Datalog, described how we evaluate programs and then motivated and described the concept of optimisation. Datalog is a declarative language for deductive databases that has a Prolog-like syntax but are evaluated using relational algebra. The most common method of evaluating Datalog programs is with a fixed-point-based algorithm (Semi-Naive evaluation) but other more complex systems have been developed. As Datalog is a declarative language the user is not required to enter programs that can be evaluated efficiently as they stand; thus optimisation is vital. In Chapter 3, semantic optimisation was motivated as being intuitive and the idea of semantic knowledge about the domain being modeled was presented. Semantic optimisation uses the domain knowledge to rewrite a program into another one that is only equivalent if it is fed data that satisfies the domain knowledge. A brief survey of some semantic optimisation systems and techniques was given. Using these foundations I analysed two well-known schemes from [CGM90, SO89]. The first is the system I labeled CGM and takes a logic-based approach using resolution and unification-type algorithms. The second, which I labeled SO, takes a relational database approach and fills some gaps in the CGM system.

The synthesis phase, Chapters 4 and 5, dealt with the design of a complete semantic optimisation system. In Chapter 4, I sketched the design of such a system which is composed of a semantic knowledge manager, a recursive component optimiser, a global

optimiser and a single rule optimiser. The last of these has been implemented fully and is a fusion of the CGM and SO systems. Only the simplest global optimiser and constraint manager have been implemented. Chapter 5 describes the implementation of the DatalogIC system as it stands. Included is a description of how DatalogIC programs are converted into SQL statements and evaluated. The final part of Chapter 5 gives some empirical results which indicate that, given some knowledge about the workings of the RDBMS backend, semantic optimisation does produce more efficient programs.

6.2 Further Work

The are two main extensions to the work presented in this thesis: extending the power of the semantic optimiser and extending the theoretical foundations of semantic optimisation. The two are connected since we cannot extend the system until we understand the effect any extension has, and this can only be achieved through theoretical tools and improvements in the methodology one uses when designing a semantic optimiser.

The first type of improvement to semantic optimisation is increasing the expressiveness of the constraints that can be handled by the optimiser. This would include the ability to handle constraints which have negation, disjunctions and functions. If the last of these can be achieved we can use Skolem functions to model existential variables and thus solve the subset constraint problem mentioned in Section 3.1.3. Related to this is the adding of these features to the program rules themselves. This is more difficult since we would have to determine the semantics of these new features and this is strongly tied up with evaluation, which is not the case with constraints. The second type of improvement we can make is with the handling of recursive programs. Recursion is a powerful modeling tool and if we are to provide an efficient declarative language, optimisation of recursive programs is required. A lot of work has been done on the evaluation and syntactic optimisation of recursive Datalog programs and the addition of semantic optimisation to this work will add another facet to each of the recursive systems.

The methodological improvements involve expanding the logical foundations, the use of higher level languages, and the possible use of mechanical theorem provers. If we are to expand the system to handle disjunctions, functions, negation and recursion, a corresponding expansion of the logical foundations will have to be carried out. A common

representation for all the many different optimisation schemes, which arise from different points of view (database, AI or logic), will enable us to compare optimisation schemes, comment on their relative performances and finally to develop a system which encompasses the good points of all. This could possibly lead to a theory of optimisation where we study the interaction between systems, such as between magic sets and semantic optimisation. All of this could use as its basis the work in [CGM90]. The use of a language like Prolog or LISP for the writing of experimental systems is preferable to using a language like C. However, as execution speed is important, there is a role for lower level languages especially if we wish to evaluate programs using a DBMS such as Oracle. A higher-level language will enable us to confirm that the implementation does actual conform to the theory. Determining this would be made easier if an interactive theorem prover were used. The ideal scenario is as follows: We would enter a theory describing Datalog and semantic knowledge into the system. We add, in some representation such as LISP or Prolog, the algorithms which describe the semantic optimisation system system plus axioms for these languages. Then we see if we can prove theorems which tell us that these algorithms preserve semantic equivalence.

Appendix A

Data Structures and Algorithms

This appendix details some of the data structures and principle algorithms used in the DatalogIC system. First, in Section A.1, I describe the basic data structures and how rule/goal graphs are represented. In Section A.2, I outline how the parser builds up the rule/goal graph. Section A.3 gives some more details on the safety and rectification algorithms, the graph module used and the algorithms for the Magic Sets optimisation. Finally, in Section A.4 I give the algorithms for the SO optimisation system, as detailed in Section 3.3.

A.1 Data Structures

In this section, I outline the main data structures used in the system, with the aid of Figures A.1, A.2, and A.3. First, I describe the individual building blocks and then I show how these are combined to hold DatalogIC rules and integrity constraints.

A.1.1 Basic Building Blocks

The data structures in the system are essentially linked lists of instances of the C `struct` construct. The main structures are as follows.

Definition Nodes

The `DEFNNODE` structure holds the information for both the intensional and extensional predicates. For intensional predicate definitions, the structure has the following major fields:

- Name of the definition.
- Argument List.
- Non-recursive rule list.
- Recursive rule list.
- Integrity constraint list—list of all ICs containing this predicate.
- Child list—list of all definitions which depend on this one.
- Recursive component—list of the other definitions in this definition's recursive component.
- Pointers to left and right children in definition tree.

Extensional definitions will have only the name, argument list and integrity constraint list fields.

Horn Clause Nodes

The Horn clause structure, `HORNCLAUSE`, is used for the rules, queries and integrity constraints. The following is a list of the major fields in the structure. An $*$ indicates fields that are used for rules and queries only, while a $+$ indicates fields that are used for integrity constraints only.

- Name of definition being defined.$*$
- Argument list for the head of rule.$*$
- Head predicate of integrity constraint.$+$
- Body predicate list.
- Relop/comparison list.

See Figure A.1 for how a rule is represented.

Predicate Nodes

The PREDNODE structure can be of two types—PREDICATE and RELOP. The former is for non-evaluable predicates, while the latter is for evaluable predicates. These are linked together to form the body of Horn clauses. For database predicates the structure is as follows:

- Name—Name of the predicate
- Argument list—List of arguments in this occurrence.
- Predicate Number—A number indicating the predicate's position in the body of the Horn clause.
- Pointer to definition for this predicate. This is the definition to rule arc in the rule/goal graph.

There is one occurrence of the structure for each occurrence of a predicate in the program. Evaluable predicates are held in a structure that has fields for:

- Type of operator ('=', '$\neq$', '>', '$\geq$', '<' or '$\leq$').
- Pointer to first operand.
- Pointer to second operand.

Argument Nodes

An argument that appears in a definition head, the head of a rule or a predicate occurrence is held in the ARGNODE structure. An argument can be one of three types—VARIABLE, CONST_STRING and CONST_NUM. The union type structure is not used since there is some sharing of fields.

- Type of argument.
- String—This is the text version of the argument (as it appeared to the lexical analyser)
- Number—For variables this is the argument's position in an argument list; for CONST_NUM arguments this is the number itself.

- Adornment—used in production of adorned program (possible values are `BOUND` or `FREE`).
- Label—used in queries (possible values are `BOUND`, `FREE`, or `EXISTENTIAL`).

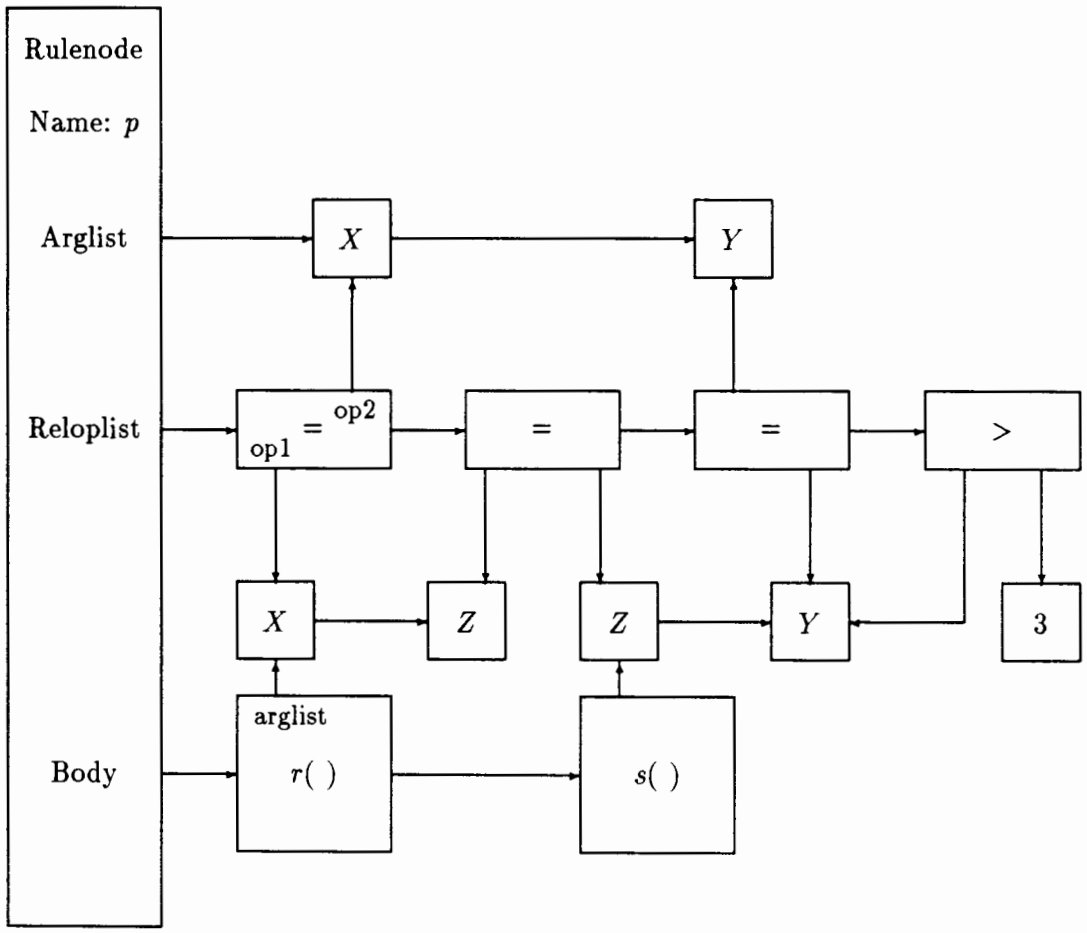
A.1.2 Representing the Rule/Goal Graph

The rule/goal graph for the rule set R is a graph, (V, E), where

- The vertex set, V, is the union of the set of rule nodes and the set of predicate nodes.
- The edge set is made up of two types of directed arcs: $p \rightarrow r$ if the rule r contains the predicate p in it's body, and $r \rightarrow p$ if r is a rule defining the predicate p.

Note that in the data structures that follow the arcs are represented by pointers going in the reverse direction. This was done since most processing is done starting with the query and not the base extensional predicates. Rules are held in the `RULENODE` structure and an example of how this is done for a fully-rectified rule is shown in Figure A.1. The `RULENODE` structure has a field for the name of the rule (which is the same as the head predicate's name) and fields pointing to three lists: the list of arguments for the rule head, the list of comparison operators (the *reloplist*) and the list of predicates that make up the body of the rule. Since the rule is fully rectified there is one variable for each argument position in the rule and any variables that occupied two positions in the original rule (such as the X in the head and the X in the body) will linked by an "=" comparison operator. Arguments are held in the `ARGNODE` structure which can hold constants (numeric and string) and variables. The body is a list of `PREDNODE` structures, one for each predicate occurrence in the rule body. The `PREDNODE` has a field for the name of the predicate, its list of arguments and a pointer to the `DEFNNODE` for the predicate (this is the definition to rule link in the rule/goal graph). Since constants in a fully-rectified rule cannot be in a predicate these are left dangling—as is the constant 3 in the figure.

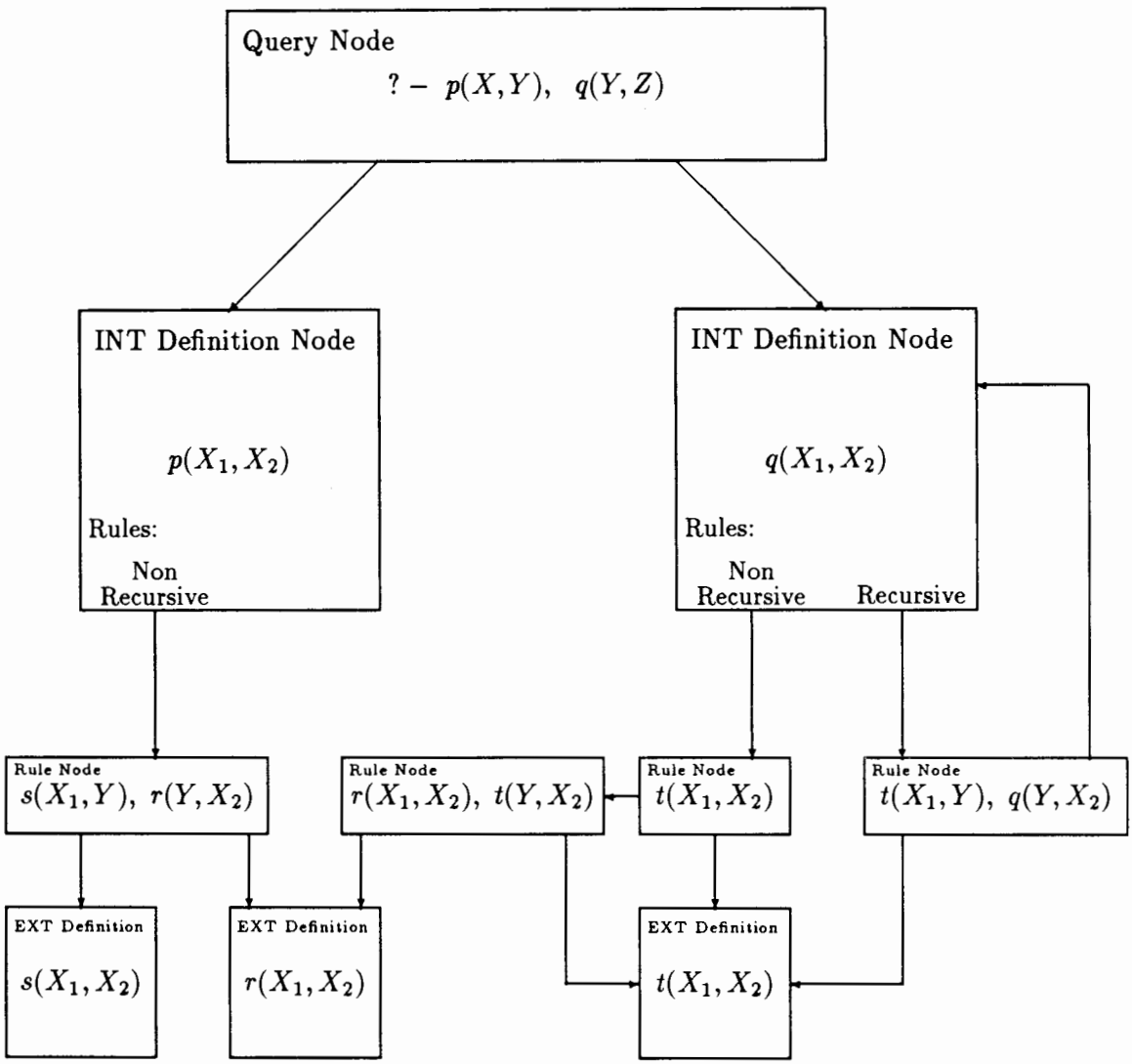
Figure A.2 shows how a program, less the semantic knowledge (see Section A.1.3 for how this is held by the constraint manager), is represented in the system. Rules and queries are both held in `RULENODE` structures while definitions in are held in the `DEFNNODE` structures. Each of these, along with their associated lists of arguments and predicates, are shown as boxes in the diagram. The overall data structure is a graph reachable from the query, with the extensional predicates as sink nodes. The $q(X_1, Y_2)$ predicate has two



Unrectified: $p(X,Y)\ :-\ r(X,Z), s(Z,Y), Y > 3$

Rectified: $p(X_0,Y_0)\ :-\ r(X_1,Z_1), s(Z_2,Y_2), X_0 = X_1, Z_1 = Z_2, Y_0 = Y_2, Y_2 > 3$

Figure A.1: Representation of a Rule



$$p(X_1, X_2) : -s(X_1, Y), r(Y, X_2).$$
$$q(X_1, X_2) : -t(X_1, Y), q(Y, X_2).$$
$$q(X_1, X_2) : -r(X_1, X_2), t(Y, X_2).$$
$$q(X_1, X_2) : -t(X_1, X_2).$$
$$? - p(X, Y), q(Y, Z).$$

Figure A.2: Representation of the Rule/Goal Graph

non-recursive rules and one recursive rule defining it, which are divided into two linked lists. Note that there is a link from the recursive rule back to the **DEFNNODE**. The $p(X_1, Y_2)$ definition has only one defining rule. There are three extensional predicates, shown at the base of the diagram.

A.1.3 Constraint Manager

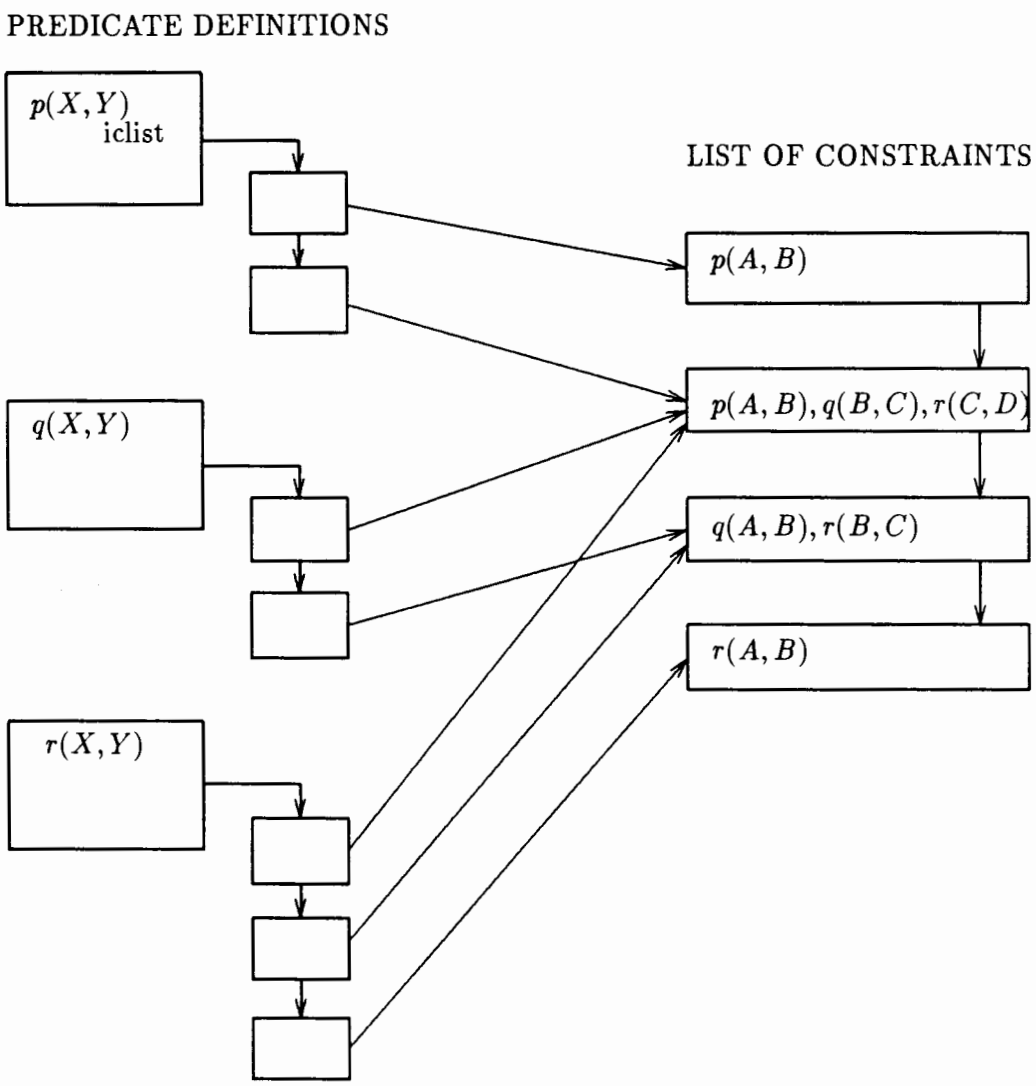


Figure A.3: Representation of Constraints in the Manager

Figure A.3 shows how constraints are held in the constraint manager. Each **DEFNNODE** points to a list of constraints (the *iclist*) which contain that predicate (to be exact a list of pointers to constraints). Thus when a set of possibly merge-compatible constraints is requested for a rule, the manager uses these lists for each predicate appearing in the rule,

ordered on the number of literals not in the rule (as outlined in Section 4.3).

A.2 The Parser

In this section, I expand on the details given in Section 5.2, and describe how the parser builds up the rule/goal graph. When the parser encounters a particular non-terminal (such as an argument or a rule) an action is carried out. Most of the actions add the structure currently being parsed to the rule/goal graph. The principal actions are as follows:

- **Build Definition Node** There are four types of predicate definition—`EXTEN`, `INTEN`, `HUNDEF`, `UNDEF`. The third and fourth handle the case where the predicate is used before its definition; the third when the predicate appears at the head of the rule, the fourth if it appears in the body of a rule or an integrity constraint.

  The first thing that the parser does when it encounters a definition in the program is to search the definition tree to see if the definition is already present. If it is not present, then a new `DEFNNODE` is built and filled; if it is present, then the type of the found definition is changed to `INTEN` or `EXTEN`. However if the node found has type `HUNDEF` and the current node has type `EXTEN`, we have an error.

- **Build Predicate Node** The definition for the predicate is searched for and if it is not found an `UNDEF` definition is created. A `PREDNODE` is then created and filled with the name of the predicate and the argument list. Finally, the number of arguments in the predicate is checked against the number in the definition.

- **Build Rule Node** A rule can be encountered outside a definition (where it must have a head) or inside a definition (where it need not have a head). In the latter case the head and its arguments are obtained from the head of the definition containing the rules. Once built, the rule is then rectified (see Section A.3.1) and the body split into a list of evaluable predicates (the reloplist) and a list of database predicates. Finally, the rule is added to the current definition's non-recursive rule list. Recall, that this list is split into non-recursive and recursive rule lists once the program has been parsed.

- **Build Relop Node** A `RELNODE` is created which is then added to the current predicate list. Since we prefer variables to be the leftmost operand we swap the operands

if the left operand is not a variable and reverse the operator.

- **Build Argument Nodes** Arguments can be of three types: `VARIABLE`,`CONST_STRING` or `CONST_NUM`. If a variable appears in a query it may have a label (? or !) indicating if it is an output or an input variable.

- **Build Query** A query is treated in the same way as a rule except that if it is written without a head then one has to be built. The argument list of the head is composed of all input and output variables in the query body.

- **Build Integrity Constraints** As with rules, these can appear inside or outside the scope of a definition. If an integrity constraint appears inside the scope of a definition it is taken to mean that the constraint refers to the definition being defined, so an occurrence of the defined predicate is added to the constraint body. If the constraint has a head which is a comparison, the head is negated and moved into the body of the constraint. As with rules, the constraint is rectified and the body divided into non-evaluable and evaluable predicates. Finally, the constraint is handed to the integrity constraint manager which will add it to the constraint list of all the definitions of predicates which have an occurrence in the constraint.

A.3 Algorithms

A.3.1 Rectification and Safety

A fully rectified Horn clause is one with no multiple occurrences of variables, and no constants in the head or the non-evaluable predicates of the body. This implies that all multiple occurrences of variables and constants will appear only in the evaluable predicates. For instance the rule

$$p(X, X, a) \ :- \ q(X, Y), r(Y, Z), Z > 3.$$

is rectified to

$$p(X_0^1, X_0^2, X_0^3) \ :- \ q(X_1^1, Y_1^2), r(Y_2^1, Z_2^2), Z_2^2 > 3, X_0^1 = X_0^2, X_0^3 = a, X_0^2 = X_1^1, Y_1^2 = Y_2^1.$$

One can see that most of the information about the rule is contained in the list of evaluable predicates. This is made use of in the conversion to SQL and in the SO part of the local optimiser. The rectification algorithm is shown in Figure A.4.

Algorithm: Fully Rectify a Horn Clause
INPUT: Horn Clause
OUTPUT: Fully Rectified Horn Clause
DATA STRUCTURES: Lookup table, T, for variables.

```
BEGIN
    For every predicate, p_i, in the rule (including head)
        For every argument a in arglist of p_i
            If a is a variable, X
                Convert X to X_j^i form.
                If there is a variable v with same name as X in T then
                    Add "v = X_j^i" to the reloplist for the rule.
                Add X_j^i to T (replacing v if necessary).
            else
                Copy and convert a to a variable x_a.
                Add "a = x_a" to reloplist.
            endif
        endfor
    endfor
END
```

Figure A.4: Rectification

A rule is safe if all the variables in the head predicate are *limited*. A variable is limited if:

- It is equated to a constant or another limited variable.
- It appears in a database (non-evaluable) predicate.

The algorithm is basically a propagation of 'limitedness' though the '=' operator from the constants and database variables, followed by a check to see if head variables are limited. This can be acheived by a `while` loop which applies the two rules above and which is exited when no change occurs.

A.3.2 Graph Module

As two separate parts of the system use graph-theoretic algorithms, a reusable graph module was created. The module consists of functions which identify strongly connected components (for recursive components in the rule/goal graph and the equivalence classes in the conjunction graph) and functions which form the transitive reduction of an acyclic graph (for transitive reduction of the conjunction graph).

The data structure used to hold a graph is a simple adjacency list. This is a list of nodes where each node has a list of those nodes to which it has arcs. The interaction with the client module is through the following functions:

- make_adj_list(num)—Sets up adjacency list with num nodes.
- add_arc(from,to)—Adds arc (from,to) to adjacency list of the from node.
- read_node()—Makes next node the current node.
- read_arc()—Returns next node in adjacency list of current node.
- find_strong_component()—Finds strongly connected components.
- transitive_reduction()—Forms transitive reduction.

The algorithm which calculates the transitive reduction is shown in Figure A.5. The algorithm to calculate the strongly connected components is quite complex and can be found in [Gib85]

A.3.3 Magic Set Optimisation

Magic set optimisation is an attempt to simulate the sideways information passing feature of a top-down evaluation algorithm like Prolog's. This is achieved by rewriting the rules so that bindings available at query time are passed down the reduced rule/goal graph by means of the one or more *magic* predicates. The algorithm to form the adorned programs is shown in Figure A.6 while Figure A.7 shows the algorithm to calculate the magic program. See Section 2.4.2 for an example of what these algorithms do.

Algorithm: Forms Transitive Reduction
INPUT: Acyclic Graph G
OUTPUT: Transitive reduction of G. All paths but the longest between any two nodes are dropped

```
BEGIN
    Sort the nodes topologically (an order based on the arcs).
    For each node, c in new node list (in reverse order)
        Add current node c to it's own reachability list.
        For each node n directly reachable from c, taken in descending order
            If n is on the reachability list for c then
                Delete the arc.
            else
                Add n to reachability list of c.
                Append the reachability list of n to that of c.
            endif
        endfor
    endfor
END
```

Figure A.5: Forming Transitive Reduction

Algorithm: Calculating the Adorned Program
INPUT: Program (including query form)
OUTPUT: Adorned Program

```
PROCEDURE Calc Adorn Pred(p ,a)
/* a is the adornment for p */
    Rename p to p^a
    Search for p^a in the new definition tree
    If not found
        Create new definition node for p^a
        Copy information from defn p to defn for p^a
        For every rule r in defn p
            Calc Adorn Rule( r, a ) and add to rule list for p^a
    endif
    return p^a
ENDPROC


PROCEDURE Propagate( r,a)
/* Propagate bindings in a through the rule r */
    Mark all head variables in bound positions as distinguished.
    Mark all variables equated to constants as distinguished.
    While some variable's status changes do
        All variables in an EDB predicate that contains at least one
            distinguished variable are distinguished.
        All variables equated to a distinguished variable are distinguished
    endwhile
ENDPROC


PROCEDURE Calc Adorn Rule( r , a)
    Propagate(r , a)
    The body of the new rule is identical to r except that
        every IDB p_i is replaced with Calc Adorn Pred( p_i ).
ENDPROC

BEGIN
    Calc Adorn Rule( query)
END
```

Figure A.6: Calculating the Adorned Program

Algorithm: Calculating the Magic Program
INPUT: Adorned Program (including query form)
OUTPUT: Magic Program

PROCEDURE Calc Magic(r)

/* r is an adorned rule of the form
$h^a(B,F) : -p_i^a(B_i,F_i),\ldots,q_j(X_j)$
where are B and F are vectors of bound and free vars,
p_i^a IDB predicates and q_j EDB predicates.*/

For every IDB predicate, $p_i^a(B_i,F_i)$, in the body build a magic rule, mr, as follows:
$magic\_p_i^a(B_i)$ is the head (if B_i empty go to next p_i^a).
$magic\_h^a(B)$ is first pred in body.
For every EDB pred, $q_j(X_j)$, in rule r
If at least one of the variables in X_j is bound then
Add $q_j(X_j)$ to body of mr.
Add mr to the rule list for $magic\_p_i^a$.
ENDPROC

BEGIN MODULE
Calc Adorn Program (query).

For every rule r in the adorned program
Calc Magic(r).

For every rule r
If the head is $p^a(B,F)$ then
Add magic predicate $magic\_p^a(B)$ to the body of r.

Using the constants in the query to build base magic rules.
END

Figure A.7: Calculating the Magic Program

A.4 Algorithms for Local Optimiser

This section gives one possible procedural presentation of the above declarative one. The following algorithms are expansions of the ones given in [SO87, SO89]. Improvements can be made to some algorithms, and these I will mention.

The first algorithm, shown in Figure A.8, determines if an edge (x, y, α), is implied by a graph. The first part searches for a path between the two nodes. If there is one, then the dominant label is calculated by first considering the case where α is $\neq$, and then when it is not.

Algorithm: Arc Implication. Determines if $G \vdash (x, y, \alpha)$.
INPUT: Condensed Conjunction graph, G and an edge (x, y, α).
OUTPUT: True if $G \vdash (x, y, \alpha)$. False otherwise.

```
BEGIN
    Find the dominant label, β, of the path from x to y.
    If α is ≠ then
        Find the dominant label, β', of the path from y to x.
        If β is ≠ or one of β' or β is > then
            Return True.
        else /* Includes case where there were no paths */
            Return False.
    else /* See if β implies α */
        If β = α then
            Return True.
        else if β is > and α is ≥ then
            Return True.
        else /* Includes case where there is no path*/
            Return False.
END
```

Figure A.8: Arc Implication

The algorithm in Figure A.9 takes the fully rectified rule and builds the conjunction graph from it. The algorithm, which is just a straight implementation of the definition, iterates through the variables and constants building the nodes, and then iterates through the comparisons operators building the edges.

To calculate the canonical condensed form the next algorithm, in Figure A.10, first builds the equivalence classes, then, using the edges from the old graph, builds the edges for the new one. Finally, the algorithm checks for consistency using the $>$ and $\neq$ edges.

Algorithm: Form conjunction graph.
INPUT: Fully Rectified Rule.
OUTPUT: Conjunction Graph $G = (V, E)$.

BEGIN
    The vertex set, V, is $X_1 \cup \ldots \cup X_n \cup C$.
    For every node equated to a head variable
        Label it as a target node.
    For every comparison, $x\ \alpha\ y$ in δ
        If $\alpha \in \{>, \neq, \geq\}$ then
            Add (x, y, α) to the edge set E
        else if α is $<$ then
            Add $(y, x, >)$ to E.
        else if α is $\leq$ then
            Add $(y, x, \geq)$ to E.
        else /* α = "=" */
            Add $(x, y, \geq)$ and $(y, x, \geq)$ to E
    For every pair of constants from the same domain
        Add the implicit edge, $(a, b, >)$ or $(b, c, >)$.
END

Figure A.9: Form Conjunction Graph

The equivalence classes are formed by a call to a standard algorithm (in the graph module) which finds the strongly connected components of the graph (on the $\geq$ edges only).

Algorithm: Form Canonical Condensed Form of a Graph
INPUT: Conjunction Graph, G
OUTPUT: Canonical Condensed Form of G, G^*

BEGIN
/* Form equivalence classes */
For the graph G find its strongly connected components.
Form G' with $E = \emptyset$ and V as the set of components of G.

/* Map the edges */
For every edge, (x, y, α), in G
Determine which nodes in G' x and y appear in (we'll call these x' and y').
If $y' = x'$ and α is $>$ or $\neq$ then
G^* is the null graph.
else if there is already an edge, $e = (x', y', \alpha')$, between x' and y' then
If $\alpha \neq \alpha'$ then Set the label of e to be $>$.
else add (x, y, α) to G'

/* Test Consistency */
For every edge $(x, y, >)$
If there is a path from y to x with dominant label $\geq$ or $>$, or if $x \equiv y$ then
G^* is the null graph.
For every edge $(x, y, \neq)$
If $x \equiv y$ then
G^* is the null graph.
END

Figure A.10: Form Canonical Condensed Graph

As with all closure type operations, the simplest implementation is to have a `while` loop which is run until no more changes take place (as in Semi-Naive evaluation). The semantic expansion algorithm, Figure A.11, is a loop composed of two parts. In the first part, edges are added from those constraints where all but one of the literals are flaged as being implied by the graph (this is done first in the loop as there may be edges that we can add without checking implication—for instance if the constraint contains just one literal). In the second part, the algorithm will test to see which literals in the body of the constraints are implied by the graph, and flag them as such. The loop is exited if no more implied edges are found or if the graph contradicts a constraint. A simple improvement

would be to test for implication only those edges that could have been affected by the adding of the new edges.

Algorithm: Form Semantic Expansion of a Graph
INPUT: Conjunction Graph G and Set of Residues I
OUTPUT: Semantic Expansion of G with respect to I

```
BEGIN
    Initialise flags in members of I.
    Repeat
        /* Add implied edges */
        For every integrity constraint, ic, in I that is still unused
            If there is only one A_i marked as not implied by G then
                Add the negation of A_i to G and mark ic as used
        Restore Condenseness Property of G (use Algorithm A.10)
        /* Find Implied Edges */
        For every integrity constraint, ic, in I that is still unused
            For every A_i
                If G ⊢ A_i then
                    Mark A_i as being implied by G.
            If all A_i of ic are implied then
                G becomes the null graph.
    Until No more implied edges are found or G becomes the null graph
END
```

Figure A.11: Semantic Expansion

The next algorithm, Figure A.12, does not have to be run within a `while` loop since the removal of an edge will not make any edge become implied by the graph that wasn't implied before (monotonicity). The first part removes edges implied by the constraints and the graph (testing $\vdash_I$), while the second part removes edges implied by the graph only (testing $\vdash$).

The algorithm to remove redundant relations, Figure A.13, is the most complex of the algorithms used in the transformations mainly because the definition is quite complex and difficult to formalise. The algorithm uses a data structure for each predicate called `clists` which keeps a record of which arguments are covered by arguments from other predicates. If the `clists` of a predicate occurrence contains all the arguments of the predicate then the predicate occurrence is redundant. The first stage is to remove predicates which are duplicated elsewhere in the graph. Avoiding the situation where a predicate is removed by

Algorithm: Removing Redundant Edges from Conjunction Graph
INPUT: Conjunction Graph G and Set of Residues I
OUTPUT: Redundant Edge Free Graph

BEGIN
For every integrity constraint, $(A_1, \ldots, A_k \rightarrow)$, in I
If there is a j such that $\forall i \neq j \; G \vdash A_i$, and $G \vdash \neg A_j$ then
Remove $\neg A_j$ from G (if it is present).

For every edge, e, in G
If $G - \{e\} \vdash e$ then
Remove e from G.
END

Figure A.12: Removing Redundant Edges

virtue of being duplicated one or more predicates which themselves are removed as being redundant. The next stage is to remove dangling relations. The algorithm follows the definition using the `clists` to mark off those arguments of a predicate occurrence which statisfy the conditions.

Figure A.14 shows the algorithm which converts the graph back to a rule. First we expand each multi-member node linking all the elements with a = operator, and then map the non-implicit edges onto comparison operators. Multi-member nodes are expanded in such a way as to use the smallest number of = operators as possible.

The complete algorithm for the SO system is merely a sequential execution of each of the algrithms as shown in Figure A.15.

Algorithm: Removing Redundant Nodes from Conjunction Graph.
INPUT: Conjunction Graph G and Set of Subset Constraints S.
OUTPUT: Redundant Node Free Graph.
DATA STRUCTURES: Clist = attribute containment list one for each predicate occurrence in the graph.

BEGIN
/* Remove Duplicates */
For every node, n, in G
For every pair of variables, v_1, v_2, in N
If v_1 and v_2 appear in occurrences of the same predicate in the same position then
Add v_1 to the Clist for the occurrence in which v_2 occurs.
Add v_2 to the Clist for the occurrence in which v_1 occurs.

For every predicate occurrence, q_i, in G
If every attribute appears in the Clist then q_i is redundant.

/* Remove Dangling Relations: Condition 2 */
For every subset constraint, $S.B \subseteq R.A$, in S
If $G \vdash (A = B)$ then
Add B to the containment list of the occurrence where A appears.

/* Condition 1 and 3 */
For every node, n, in G
If n appears in no edges, n is not target and the set of variables
attached to n contains only one element v then
Add v to the containment list of the occurrence it appears in.

For every predicate occurrence, q_i, in G
If every attribute appears in the Clist then q_i is redundant.
END

Figure A.13: Removing Redundant Nodes

Algorithm: Conversion of Graph Back to Rule Form
INPUT: Conjunction Graph (Possibly Condensed), G
OUTPUT: Fully Recitified Rule

BEGIN
Start off with the original rule (less its δ part).
For every node, n, in G (which has representative r)
For every member, v, of the variable/constant set attached to n
Add '$r = v$' to δ.
For every edge, (x, y, α), in G
If x and y are not both constants
Add '$x\ \alpha\ y$' to δ.
Drop those q_i which do not have variables in δ.
Return the rule.
END

Figure A.14: Conversion Back to Rule Form

Algorithm: Local Optimisation
INPUT: Fully Rectified Rule, R, Residues, I and Subset Constraints, S
OUTPUT: Optimised Rule, R'

BEGIN
G_1 = Form conjunction graph (R).
G_2 = Form canonical condensed graph (G_1).
G_3 = Form semantic expansion (G_2).
G_4 = Remove Redundancies (G_3).
R' = Convert back to rule form (G_4).
END

Figure A.15: Using Residues

Bibliography

[ASU79] A.V. Aho, Y. Sagiv, and J.D. Ullman. Equivalences among relational expressions. *SIAM Journal of Computing*, 8:218–246, 1979.

[BJ86] M. L. Brodie and M. Jarke. On integrating logic programming and databases. In L. Kerschberg, editor, *Expert Database Systems: Proc. From the First Int. Workshop*, pages 191–207. Benjamin-Cummings, Menlo Park, CA, 1986.

[BMSU86] F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proc. 5th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pages 1–15, 1986.

[BR86] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proc. ACM-SIGMOD Conference on the Management of Data*, pages 16–52, 1986.

[BR87] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proc. 6th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 269–283, 1987.

[CFM86] U.S. Chakravarthy, D.H. Fishman, and J. Minker. Semantic query optimization in expert systems and database systems. In L.Kerschberg, editor, *Expert Database Systems: Proc. From the First Int. Workshop*, pages 659–675. Benjamin-Cummings, 1986.

[CGM87] U.S. Chakravarthy, J. Grant, and J. Minker. Semantic query optimization: Additional constraints and control strategies. In L. Kerschberg, editor, *Expert Database Systems: Proc. From the First Int. Conference*, pages 345–379. Benjamin-Cummings, Menlo Park, CA, 1987.

[CGM88] U.S. Chakravarthy, J. Grant, and J. Minker. Foundations of semantic query optimization. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 659–698. Morgan Kaufmann, Los Altos CA, 1988.

[CGM90] U.S. Chakravarthy, J. Grant, and J. Minker. Logic based approach to semantic query optimization. *ACM Transactions of Database Systems*, 15(2):162–207, June 1990.

[CGT89] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–167, 1989.

[CGT90] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Springer-Verlag, New York, 1990.

[Gib85] A. Gibbons. *Algorithmic Graph Theory.* Cambridge University Press, 1985.

[GMN84] H. Gallaire, J. Minker, and J.M. Nicolas. Logic and databases: A deductive approach. *ACM Computing Surveys*, 16:153–185, June 1984.

[HH87] J. Han and L. Henschen. Handling redundency in the processing of recursive database queries. In *Proc. ACM-SIGMOD Conference on the Management of Data*, pages 73–81, 1987.

[HL88] J. Han and S. Lee. Semantic query optimisation in recursive databases. In *Proceedings of 4th International Conference on Data Engineering*, pages 444–451, 1988.

[HM75] M. M. Hammer and D.J. McLeod. Semantic integrity in relational database systems. In *Proc. of the First Int. Conference on Very Large Data Bases*, pages 25–47, 1975.

[HN84] L.J. Henschen and S.A. Naqvi. On compiling queries in first order databases. *ACM Computing Surveys*, 31(1):11–155, Jan 1984.

[Jar86] M. Jarke. External semantic query simplification: A graph theoretic approach and its implementation in Prolog. In *Expert Database Systems: Proc. From the First Int. Workshop*, 1986.

[JCV84] M. Jarke, J. Clifford, and Y. Vassiliou. An optimising Prolog front-end to a relational query system. In *Proc. ACM-SIGMOD Conference on the Management of Data*, pages 296–306, 1984.

[JK84] M. Jarke and J. Koch. Query optimisation in database systems. *ACM Computing Surveys*, 16(2):111–155, June 1984.

[Kin81] J. J. King. Quist: A system for semantic query optimisation in relational databases. In *Proc. of the Seventh Int. Conference on Very Large Data Bases*, 1981.

[KL85] M. Kifer and E. Lozinski. Query optimisation in logical databases. Technical Report 85/16, State University of New York at Stony Brook, June 1985.

[KP82] A. Klug and P. Price. Determining view dependencies using tableaux. *ACM Transactions on Database Systems*, 5(3):361–380, September 1982.

[Llo87] J. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

[Mah88] M. J. Maher. Equivalence of logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 627–658. Morgan Kaufmann, Los Altos CA, 1988.

[Meh84] K. Mehlhorn. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness.* EATCS Monographs in Theoretical Computer Science, Springer-Verlag, Berlin, 1984.

[MFP90] I.S. Mumick, S.J. Finkelstein, and H. Pirahesh. Magic conditions. In *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Databases Systems*, pages 314–330, 1990.

[MW88] D. Maier and D.S. Warren. *Computing with Logic.* Benjamin-Cummings, Menlo Park, CA, 1988.

[MZ87] C. Malley and S. Zdonik. A knowledge based approach to query optimisation. In L.Kershberg, editor, *Expert Database Systems: Proc. From the First Int. Conference*, pages 329–344. Benjamin-Cummings, 1987.

[Nau86] J. Naughton. Data independent recursion in deductive databases. In *Proc. 5th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, 1986.

[Nau87] J. Naughton. One-sided recursions. In *Proc. 6th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 340–348, 1987.

[Ram88] R. Ramakrishnan. Magic templates: A spellbinding approach to logic programs. In *Proc. Int. Conf. on Logic Programming*, pages 140–159, 1988.

[Rei78] R. Reiter. On closed world databases. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 55–76. Plenum Press, 1978.

[Rei84] R. Reiter. Towards a logical reconstruction of relational database theory. In Brodie, Mylopoulos, and Schmidt, editors, *On Conceptual Modelling*, pages 191–233. Springer-Verlag, 1984.

[Sag88] Y. Sagiv. Optimizing Datalog programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 659–698. Morgan Kaufmann, Los Altos CA, 1988.

[She88] J. Shepherdson. Negation in logic programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 19–88. Morgan Kaufmann, Los Altos CA, 1988.

[SO87] S.T. Shenoy and Z.M. Ozsoyoglu. A system for semantic query optimisation. In *Proc. ACM-SIGMOD Conference on the Management of Data*, pages 181–195, 1987.

[SO89] S.T. Shenoy and Z.M. Ozsoyoglu. Design and implementation of a semantic query optimizer. *IEEE Transactions on Knowledge and Data Engineering*, 1(3), September 1989.

[SZ87] D. Sacca and C. Zaniolo. Magic counting methods. In *Proc. ACM-SIGMOD Conference on the Management of Data*, pages 49–59, 1987.

[Ull82] J.D. Ullman. *Principles of Database Systems.* Computer Science Press, Potomac, Md., 1982.

[Ull85] J.D. Ullman. Implementation of logical query languages for databases. *ACM Trans. on Database Systems*, 10(3):289–321, September 1985.

[Ull88] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, Potomac, Md., 1988.

[Zan86] C. Zaniolo. Prolog: A database language for all seasons. In *Expert Database Systems: Proc. From the First Int. Workshop*, pages 219–232, 1986.

Index